Towards dynamic threading support for OpenMP

Master Thesis

Author(s):
Stadler, Jacques

Publication date:
2009

Permanent link:
https://doi.org/10.3929/ethz-a-005747649

Master Thesis

Towards Dynamic Threading Support for OpenMP

Jacques Stadler

Albert Noll
Responsible assistant

Prof. Thomas R. Gross
Laboratory for Software Technology
ETH Zurich

January 2009

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Laboratory for Software Technology

Abstract

Recent developments in microprocessor design show a clear trend towards multi-core and multi-processor architectures. This radical shift in processor design results from diminishing returns of e.g. increasing processor frequencies or deeper pipelines. To exploit the available hardware resources of modern processors, programmers must write parallel code by e.g. distributing workloads to multiple threads of execution. To simplify this task, numerous approaches have been proposed. One successful candidate is OpenMP, which is a standard that provides a high-level interface for parallel programming.

While OpenMP performs well with regular workloads, unbalanced workloads can lead to inefficient resource utilization. The main reason for this inefficiency is, that the number of threads in a parallel region must remain constant throughout the parallel region's scope. As a consequence, idle resources cannot be used to assist parallel regions, even if the regions could profit from additional resources.

To remedy this issue, we propose *dynamic threading* as an extension to the OpenMP standard which allows idle threads to join active work-sharing constructs. The proposed approach was implemented in the GNU Compiler Collection. Preliminary benchmarking shows that our approach does not introduce additional overhead, and can lead to performance improvements for several scenarios.

Zusammenfassung

Neuste Entwicklungen im Microprossesor Design zeigen klare Trends in Richtung von Multicore und Multiprozessor Architekturen. Dieser fundamentale Wechsel im Prozessor Design ergibt sich aus den abnehmenden Erträgen durch das erhöhen der Prozessortaktraten und der Vergrösserung der Pipelines. Um die verfügbaren Resourcen von modernen Prozessoren auszunutzen, müssen Programmierer nun parallelen Code schreiben, zum Beispiel indem sie die Arbeitslast auf mehrere Ausführungsstränge verteilen. Um diese Aufgabe zu vereinfachen wurden eine Vielzahl von Ansätzen vorgeschlagen. Ein erfolgreicher Kandidat stellt der OpenMP Standard dar, der eine high-level Schnittstelle zum parallelen programmieren anbietet.

Während OpenMP gute Leistungen erbringt wenn es sich um ausgeglichene Arbeitslasten handelt, so kann es bei unausgeglichenen Arbeitslasten schnell zu uneffizienter Resourcenauslastung führen. Der Hauptgrund für diese Uneffizienz liegt darin, dass die Anzahl der Threads innerhalb einer parallelen Region konstant bleiben muss. Als Folge davon, können freie Resourcen, auch wenn es nützlich wäre, nicht genutzt werden um einer parallel Regionen mitzuhelfen.

Um dieses Problem anzugehen schlagen wir *dynamisches Threading* als Erweiterung zum OpenMP Standard vor, welches freien Threads das beitreten von aktiven, arbietsteilenden Regionen erlaubt. Wir haben unseren Ansatz in der GNU Compiler Collection implementiert. Erste Benchmarks zeigen dass unser Ansatz keinen zusätzlichen Overhead einführt und bei verschiedenen Fällen zu Leistungssteigerungen führen kann.

Contents

List of Figures

List of Tables

Listings

1 Introduction

1.1 Motivation

Moore's Law[22] states that the number of transistors that can be inexpensively placed on an integrated circuit is increasing exponentially, doubling approximately every two years. This statement has proven to be true by now, but in the last few years something has changed in the way the available transistors are used.

Herb Sutter, a popular C++ Expert, mentioned: "The free lunch is over"[27]. Today, to make processors faster, simply increasing the clock speeds of the processors, as e.g. Intel tried with the Pentium 4, is not sufficient anymore. As a consequence, Intel canceled the planed successors of the Pentium 4 with Prescott core in 2004, called Tejas and Jayhawk. This introduced Intel's shift from single-core to multi-core processors. The decision was mainly based on the following physical limits in processor design:

The Memory Wall stands for the ever increasing gap that spreads between the processor speeds and memory access times. Main memory accesses can lead to large latencies (several hundred cycles) in case of a cache miss. Meanwhile the processor stalls, which delays execution. To reduce the memory access latency processor designers to add bigger on-chip caches and out of order execution in order to hide this latency.

The Frequency Wall refers to the fact that instruction pipeline sizes reache performance limits. A further increase of the size of instruction pipelines results in diminishing returns regarding processor performance and even negative returns regarding power consumption.

The Power Wall denotes the increasing limitation of the processor performance by the achievable power dissipation rather than by the number of available transistors. As consequence, processor designers must improve the power efficiency of processors at about the same rate as the performance increase.

The physical limits of processor design forced architects to put more cores on a die instead of bigger caches or wider pipelines. The additional cores provide a way to potentially execute more instructions in the same time as compared to optimized single-core processors. The consequence of this trend is that application developers are now confronted with parallel programming techniques, which were relieved to be applied by system programmers before.

Writing parallel programs can be done using various approaches: A first approach is to divide the program into multiple processes, each running concurrently with a separate address space. However, inter-process communication can be expensive regarding execution time and cumbersome and error-prone to implement.

A more sophisticated approach to share data among multiple processes, without having to deal with intricate, low level communication mechanisms, is the Message Passing Interface (MPI)[2], which provides a standardized interface for communication between processes. MPI is a good approach for communication between distributed memory applications. A disadvantage of MPI is that programmers are mostly required to rewrite an application from scratch.

For symmetric multiprocessing (SMP), shared memory machines, there exist more efficient and intuitive approaches to parallelize an application. First, the use of threads instead of processes can speed up execution, since threads share a common address space, which is used to communicate more efficiently.

Although several threading libraries exist, such as the POSIX threads[7], implementing a parallel program can be tedious and result in data races or deadlocks.

Therefore, a group of hardware and software vendors defined the OpenMP standard[5] to take the burden from software developers to deal with low-level threading constructs. The OpenMP standard allows existing C, C++ and Fortran programs to be parallelized by inserting well defined pragmas that annotate code regions that are executed in parallel. The compiler and runtime system handle the low-level implementation of the parallelization.

1.2 Problem

OpenMP performs well for non-nested parallel regions, however nested parallel regions potentially perform worse because idle execution units might not be used. The reason for potentially leaving resources idle, is the fork-join execution model of OpenMP that requires the number of threads to remain constant for a parallel region, once the region was started. Irregular or input dependent workloads in nested regions can cause certain threads to be finished earlier than others. The early finished threads will stay idle, as the threads cannot join a parallel region whose number of threads must remain constant.

In this thesis an extension to the OpenMP standard is proposed, which is called *dynamic threading*, and enables the sharing of threads between teams. To achieve that threads are shared between teams, a new construct is introduced that allows OpenMP work-sharing constructs to give idle threads to other OpenMP work-sharing constructs that potentially profit from additional worker-threads. Thread sharing can in certain cases improve performance of OpenMP applications drastically while in other cases performance can also decrease, compared to the original OpenMP application. To overcome the performance penalties, extensions to the current implementation are proposed, that should further improve performance.

To introduce the reader to the subject, an introduction to OpenMP and the corresponding implementation in the GNU Compiler Collection is described in chapter 2. Chapter 3 discusses the design. Furthermore, in chapter 4 the implementation of the proposed solution is described. To give an impression of the achievements chapter 5 depicts several benchmarks and discusses the results. Chapter 6 presents related work and chapter 7 concludes the thesis.

2 Background

This chapter describes the OpenMP standard in more detail along with the OpenMP history and execution model. Additionally, the implementation of OpenMP in the GNU Compiler Collection is described. In particular, the implementation of the front and middle end and the GNU OpenMP runtime library is investigated.

2.1 The OpenMP Standard

OpenMP, which stands for Open Multiprocessing, was published by the OpenMP Architecture Review Board, a group of major hardware and software vendors. OpenMP's main aim is to simplify the programming of multi-platform, shared-memory, parallel programs by allowing insertions of pragmas into C/C++ code or special comments in Fortran code to instruct the compiler which parts of the code should be parallelized. There exist several compilers that support OpenMP, as e.g. the GNU Compiler Collection or the Intel compilers.

The OpenMP specification[5] describes compiler directives, runtime library routines and environment variables. Subsequently, a short overview over the various OpenMP constructs is given, by occasionally making citations from the specifications.

The compiler directives have a well defined syntax, as shown in Listing 2.1 for C/C++ and in Listing 2.2 for Fortran and can be divided into parallelization, work-sharing, synchronization and data sharing constructs.

Listing 2.1: Syntax of an OpenMP directive for C/C++.

```
#pragma omp directive-name [clause[ [,] clause]...] new-line
```

Listing 2.2: Syntax of an OpenMP directive for (free form) Fortran.

```
!$omp directive-name [clause[[,] clause]...]
```

- To have a region of code executed by multiple threads, the standard defines the `parallel` construct that marks the beginning of a parallel region, which encloses the following structured block of code. Each parallel region is executed by a team of threads that consists of the *master thread*, which initiated the parallel region, and zero or more additional *worker-threads*. At the

end of every `parallel` region an implicit barrier guarantees that all threads actually finished the parallel region.

- Work-sharing constructs are: loop, `sections`, `single` and for Fortran `workshare` directives. The loop directive name depends on the source language (`for` in case of C/C++ or `do` in case of Fortran).

  - The loop constructs distribute iterations of the corresponding loop among the threads of the team. The way the iterations are divided among the threads can be specified by the `schedule` clause. The different schedule kinds are described in Table 2.1, which is an excerpt from the OpenMP specification. At the end of each loop there is an implicit barrier unless the `nowait` clause is specified.
  - The `sections` construct divides a set of structured blocks (each surrounded by `section` constructs) among the threads of the team. Each `section` is executed once by one of the threads in the team. The way the `section` constructs are distributed among the threads is implementation defined.
  - The `single` construct specifies that the associated structured block is executed by only one thread of the team, which must not necessarily be the master thread. The threads of the team that are not executing the structured block wait at the implicit barrier at the end of the `single` construct, unless a `nowait` clause is specified.
  - In the Fortran language, the `workshare` construct divides the execution of the enclosed structured block into separate units of work, and causes the threads of the team to share the work such that each unit is executed only once.

  Parallelization and work-sharing constructs can also be combined by immediately nesting work-sharing constructs in `parallel` regions. Work-sharing constructs that are immediately nested in a parallel region are called *combined parallel regions* and can be abbreviated: Loops can be abbreviated as `parallel for` in case of C/C++, respectively `parallel do` in case of Fortran, and the `sections` construct can be abbreviated as `parallel sections` directive.

- The OpenMP standard provides the following synchronization constructs: `master`, `critical`, `barrier`, `atomic`, `flush` and `ordered`.

  - The `master` construct specifies a region that can only be executed by the master thread of the current team.
  - The `critical` construct restricts a structured block to be executed by a single thread at a time.
  - The `barrier` construct specifies an explicit barrier at the point at which the construct appears.
  - The `atomic` construct ensures that a given storage location is updated atomically.
  - The `flush` construct makes a thread's temporary view of memory consistent with main memory.

  - The `ordered` construct ensures that a set of instructions (surrounded by an `ordered` construct) in a loop is executed in the same order as the instructions would be when executed sequentially.

- To define a variable's scope in a parallel region, the `default`, `shared`, `private`, `firstprivate`, `lastprivate`, `reduction`, `threadprivate`, `copyin` and `copyprivate` constructs are provided by the standard.
  - The `default` construct defines the default data-sharing attribute of a variable.
  - The `shared` construct defines variables to be shared by all threads of the team.
  - The `private` construct defines variables to be private to each thread of the team.
  - The `firstprivate` construct defines variables to be private to each thread of the team and the variable to be initialized with the value the variable had when encountering the construct.
  - The `lastprivate` construct defines a variable to be private to each thread of the team and causes the original variable to updated at the end of the construct.
  - The `reduction` construct specifies an operator and a variable. Each thread will create a newly initialized, private variable which is used to do the operations in the parallel region. At the end of the parallel region the values of the private variables and the operator are used to calculate the final value and update the original variable.
  - The `threadprivate` directive specifies a global-lifetime variable to be replicated, such that each thread has its own copy.
  - The `copyin` construct copies the specified threadprivate variable from the master thread to the threadprivate variable of all threads of the team.
  - The `copyprivate` construct is used to broadcast a private variable from one thread to all thread of the team.

Furthermore, the OpenMP standard defines four environment variables, that enable compiled OpenMP programs to be configured at runtime.

OMP_SCHEDULE modifies the scheduling behavior of work-sharing loops with schedule kind `runtime`. OMP_SCHEDULE allows all scheduling variants shown in Table 2.1, except for `runtime` itself.

OMP_NUM_THREADS sets the number of threads that are used for a parallel region, in case the `num_threads` clause is not specified.

OMP_DYNAMIC specifies whether the number of worker-threads for parallel regions are dynamically adjusted to optimize the use of system resources.

OMP_NESTED enables/disables nested parallelism.

To get an overview over the runtime library routines, the reader is encouraged to have a look at the OpenMP specification.

Kind	Description
`static`	When `schedule(static`, chunk_size) is specified, iterations are divided into chunks of size chunk_size, and the chunks are statically assigned to threads in the team in a round-robin fashion in the order of the thread number. Note that the last chunk to be assigned may have a smaller number of iterations. When no chunk_size is specified, the iteration space is divided into chunks which are approximately equal in size, and each thread is assigned at most one chunk.
`dynamic`	When `schedule(dynamic`, chunk_size) is specified, the iterations are assigned to threads in chunks as the threads request them. The thread executes the chunk of iterations, then requests another chunk, until no chunks remain to be assigned. Each chunk contains chunk_size iterations, except for the last chunk to be assigned, which may have fewer iterations. When no chunk_size is specified, it defaults to 1.
`guided`	When `schedule(guided`, chunk_size) is specified, the iterations are assigned to threads in chunks as the threads request them. The thread executes the chunk of iterations, then requests another chunk, until no chunks remain to be assigned. For a chunk_size of 1, the size of each chunk is proportional to the number of unassigned iterations divided by the number of threads, decreasing to 1. For a chunk_size with value k (greater than 1), the size of each chunk is determined in the same way with the restriction that the chunks do not contain fewer than k iterations (except for the last chunk to be assigned, which may have fewer than k iterations). When no chunk_size is specified, it defaults to 1.
`runtime`	When `schedule(runtime)` is specified, the decision regarding scheduling is deferred until run time, and the schedule and chunk size are taken from the run-sched-var control variable.

Table 2.1: `schedule` clause kind values

2.1.1 History

The first OpenMP standard, OpenMP for Fortran 1.0, was published in October 1997. A year later the OpenMP Architecture Review Board released the corresponding C/C++ standard.
In 2000 version 2.0 of the Fortran specification was released, which was soon followed by the 2.0 specification for C/C++. The C/C++ and the Fortran version were improved by additional clauses, clarifications and fixes. E.g., the `num_threads` and the `copyprivate` clauses were added.
In 2005, a new version of the specification was released with the main purpose to merge the existing Fortran and C/C++ specification into a single specification with the version 2.5.
The currently latest specification of the standard was released in May 2008 with the version 3.0 and experienced several modification, amongst others a new construct called tasks.

2.1.2 Execution Model

OpenMP uses the fork-join execution model for parallel execution. In the fork-join execution model, a master thread that encounters a parallel construct forks a team of worker-threads. The worker-threads process the *parallel region* and join the master thread at the end of the parallel region. An example of a parallel region is shown in Figure 2.1.

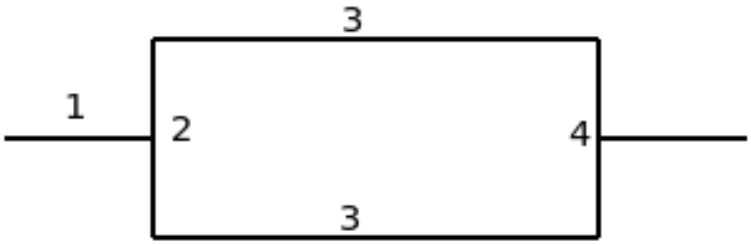

Figure 2.1: A parallel region. 1. The master thread executes a sequential region until a parallel directive is encountered. 2. The master thread forks a team with one new worker-thread. 3. Each thread executes the parallel region and waits at the implicit barrier at the end of the parallel region. 4. All threads have reached the end of the parallel region and the worker-thread joins the master thread, which can then continue executing the following sequential region.

A special case of parallel regions are *nested parallel regions*. Nested parallel regions are parallel regions that are nested in other parallel regions. The OpenMP standard does not impose a limitation to the possible depth of nesting, even though in practice having more teams that execution units does not make sense. An example of a nested parallel region is provided in Figure 2.2.

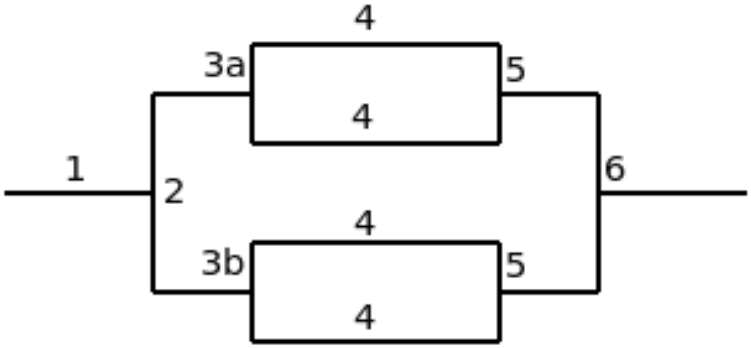

Figure 2.2: A nested parallel region. 1. The master thread executes a sequential region until a parallel directive is encountered. 2. The master thread forks a team with one new worker-thread. 3. The master thread (a) and the worker-thread (b) execute the parallel region until the second parallel directive is reached and for each thread a new team, with one new worker-thread, is created. 4. There are 3 teams, one that was created at (2), one that was created at (3a) and the third that was created at (3b). The worker-thread of the first team has become the master thread of the team created at (3b). 5. After each thread has executed the nested parallel region and reached the implicit barrier at the end of the corresponding parallel regions, the worker-threads of the inner teams join with the master threads. 6. Finally, when also the outer master and worker-thread have reached the barrier of the outer parallel region, the worker-thread joins the master thread, which can then continue executing the following sequential region.

Besides parallel regions there also exist *work-sharing regions*, which share work of a region with threads of the team. Work-sharing regions can also appear in nested parallel regions. The way the work is shared among the threads depends on the kind of work-sharing region and is described in Section 2.1. An example of a work-sharing region is provided in Figure 2.3.

Figure 2.3: A parallel work-sharing region. 1. The master thread executes a sequential region until a parallel work-sharing directive is encountered. 2. The master thread forks a team of worker-thread. 3. Each thread concurrently executes a part of the whole work-sharing region. In case of a work-sharing loop each thread would execute a part of the iterations, whereas for the `sections` work-sharing construct each thread would execute a couple of `section` constructs. 4. When all threads have reached the barrier at the end of the parallel work-sharing region, the worker-threads join the master thread, which then continues sequential execution.

2.2 OpenMP in the GCC

The GNU Compiler Collection supports the OpenMP specification 2.5 since the GCC version 4.2. Support for the current OpenMP specification 3.0 is planned for the upcoming GCC version 4.4.

Although GCC provides OpenMP support for C, C++ and Fortran, subsequent code listings are written in C. The reason is, that the implementation of OpenMP for C, C++ and Fortran differ only in the front end.

To enable the compiler to process OpenMP pragmas the `-fopenmp` compiler flag must be set. The setting of the `-fopenmp` flag will dynamically link the program against the GNU OpenMP library (libGOMP)[1] and instructs the compiler to recognize OpenMP pragmas rather than ignoring them.

To give the reader an impression of how an OpenMP program looks like, a simple example is provided in the Appendix A.1.

This thesis will concentrate on the implementation of OpenMP in GCC 4.2.4. Since the design and implementation of dynamic threading in OpenMP requires a detailed understanding of the OpenMP implementation, the subsequent sections describe how OpenMP constructs are produced by the compiler. Finally the GNU OpenMP runtime library is described shortly.

2.2.1 Front and Middle End

All OpenMP related GCC code, that is used at application compile time, resides in the GCC front and middle end. Therefore, the various back ends do not need to be aware of OpenMP semantics.

The main OpenMP specific task in the front end is to parse the OpenMP directives and clauses, perform sanity checks and propagate the directives to the middle end in the GENERIC intermediate representation (IR), described in [21]. The parsing and propagation of the directives is done in the files `c-parser.c:c_parser_omp_*`, `cp/parser.c:cp_parser_omp_*` and `fortran/parse.c:parse_omp_*` for the C, C++ and Fortran front ends.

In a next step the GENERIC representation of the code is transformed into the GIMPLE intermediate representation. The transformation is done in `gimplify.c:gimplify_omp_*` and `gimplify.c:omp_*`. During gimplification all implicit data sharing clauses are made explicit and `atomic` directives are transformed into the corresponding atomic update functions.

Passing of variables to a parallel region is implemented by a special data structure that contains space for all non-global variables that are used in the parallel region. The creation and filling of the *data-sharing data structure* is implemented in `pass_lower_omp` in `omp-low.c`. The code will also be linearized by inserting OMP_RETURN and OMP_CONTINUE instructions to denote the end of a parallel or work-sharing region.

After further intermediate passes the OpenMP expansion pass is performed. The intermediate passes are responsible for creating the control flow graph (CFG).

Before the code is put into static single assignment (SSA) form, `pass_expand_omp` is executed (in `omp-low.c`). The expansion pass outlines the body of a parallel region into a separate function and transforms other directives into the corresponding `libGOMP` calls or GIMPLE expansions.

The lowering and expansion passes will be described in detail in Section 4.3.

A rough overview over the various GCC passes can be found in Figure 2.4. Passes that are influenced by the OpenMP implementation are highlighted in yellow.

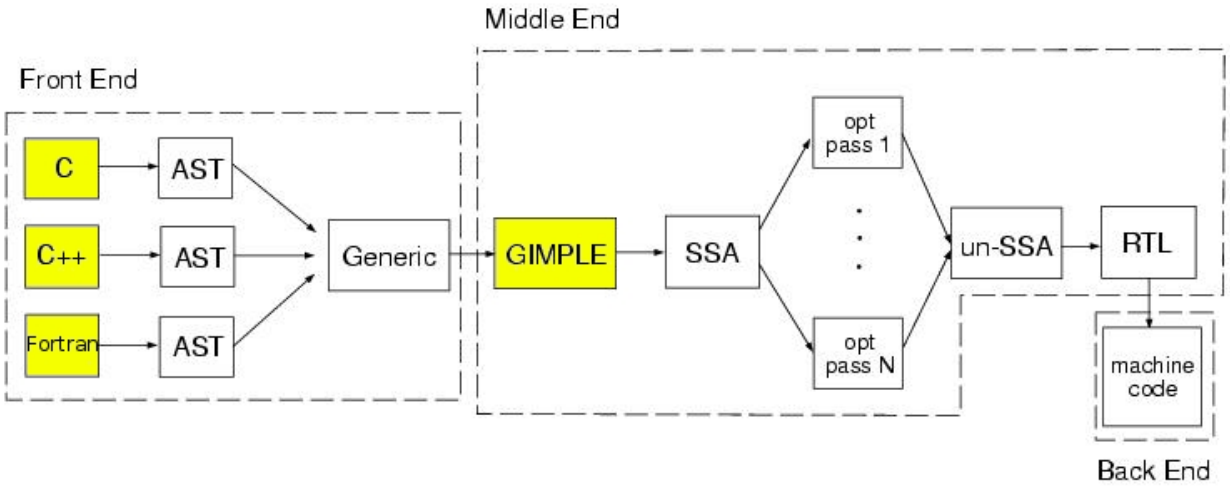

Figure 2.4: An overview of the GCC. The main OpenMP related parts are highlighted in yellow and reside in the parsers for C, C++ and Fortran and at the GIMPLE IR level.

A detailed description of the implementation of OpenMP in GCC can be found in [24].

2.2.1.1 Examples

The following listings illustrate the lowering processes performed by GCC and the used data structures.

Parallel Regions An example of the most important OpenMP directive, `#pragma omp parallel`, is shown in Listing 2.3.

Listing 2.3: Example of a parallel region.

```
...
a = 0;
#pragma omp parallel shared(a)
{
  ...
  do_something(a);
  ...
}
...
```

The compiler transforms the above code to the GIMPLE representation shown in Listings 2.4 and 2.5.

Listing 2.4: Example of the caller side of the lowered parallel region.

```
...
a = 0;
data.a = a;
gomp_parallel_start (omp_fn, &data, threads);
omp_fn (&data);
gomp_parallel_end ();
...
```

Listing 2.5: Example of the callee side of the lowered parallel region.

```
omp_fn (data)
{
  ...
  do_something(data->a);
  ...
}
```

Listings 2.4 shows the caller code that is executed by the master thread only, which fills the data-sharing struct (`data`), described in 2.2.1. Furthermore, the caller code calls the libGOMP function

to start the parallel region (`gomp_parallel_start`), then executes the parallel region and finally calls the function to end the parallel region (`gomp_parallel_end`). The number of threads for the parallel region is indicated by the `threads` parameter. If the value is zero, the concrete number is determined at runtime.
Listings 2.5 shows the outlined function (`omp_fn`) that is called with the data-sharing struct (`data`) by every thread that enters the parallel region.

Loops An example of a dynamically scheduled OpenMP loop is illustrated in Listing 2.6.

Listing 2.6: Example of a work-sharing loop region.

```
...
#pragma omp for schedule(dynamic)
for (i=0; i<10; i++) {
  /* Loop body */
}
...
```

The example code is translated into the code shown in Listing 2.7.

Listing 2.7: Example of a lowered work-sharing loop region.

```
...
gotchunk = gomp_loop_dynamic_start (0 /*low*/, 10 /*high*/,
          1 /*incr*/, 1 /*chunk_size*/, &start, &end);
if (gotchunk) goto <L1>; else goto <L4>;

<L1>:;
i = start;
limit = end;

<L2>:;
/* Loop body */
i = i + 1;
gotiter = i < limit;
if (gotiter) goto <L2>; else goto <L3>;

<L3>:;
gotchunk = gomp_loop_dynamic_next (&start, &end);
if (gotchunk) goto <L1>; else goto <L4>;

<L4>:;
gomp_loop_end ();
...
```

Before the loop is entered, `gomp_loop_dynamic_start` is called. `gomp_loop_dynamic_start` is used to initialize the dynamically scheduled loop work-sharing construct and to retrieve a chunk of loop iterations for the current thread to execute. After having executed a chunk, the current thread will call `gomp_loop_dynamic_next` to retrieve a new chunk of iterations until there are none left. If there is no chunk left, `gomp_loop_dynamic_start` and `gomp_loop_dynamic_next` will return zero and the current thread exits the loop by calling `gomp_loop_end`.

2.2.2 OpenMP Runtime Library - libGOMP

As described in the previous section, the compiler generates calls to an external library, the libGOMP. In particular, the libGOMP is responsible for thread management, as well as the distribution of loop iterations and sections to requesting threads. For the management of threads, libGOMP uses the POSIX threads (Pthreads) library [7], which provides a standardized API for creation and manipulation of threads and is available across many architectures. To manage the threads, libGOMP makes use of several structs: `gomp_thread`, `gomp_team_state`, `gomp_team` and `gomp_work_share`.

- The `gomp_thread` is the libGOMP abstraction for a thread, and mainly consists of a pointer to the function that is currently executed, a pointer to the data-sharing struct used in this function, a semaphore used for ordering of loop iterations and a `gomp_team_state` in order to know which team the current thread is associated to.
- The `gomp_team_state` is associated to every thread, and stores pointers to the `gomp_team` and `gomp_work_share` structs along with a team id that uniquely identifies a thread within a team.
- The `gomp_team` is the abstraction of a team in libGOMP and provides an array of pointers to `gomp_work_share`s of all the work-sharing regions that are currently active within the team and a lock to ensure that access to the team happens mutually exclusive. Furthermore, `gomp_team` provides a barrier for team synchronization, the team size, the previous `gomp_team_state` (before the master thread entered the current team) and other structures to ensure the management of the active work-sharing regions.
- The `gomp_work_share` provides the abstraction of a work-sharing region, which stores the schedule kind of loops along with the next and last loop iteration variable value as well as the chunk size and the step size, by which the loop iteration variable is increased after an iteration. To ensure exclusive access by multiple threads, a lock is provided. To determine whether a work-sharing region can be deleted, `gomp_work_share` also needs to manage a variable that stores the number of threads that have left the current work-sharing region. Finally, to enable the ordered clause to be implemented further variables are provided.

The libGOMP structs are illustrated in more detail in Listing A.3 of the Appendix.

Furthermore, libGOMP is responsible for mapping the OpenMP runtime library routines that require synchronization or timers to the underlaying operating system primitives.

2.2.2.1 Thread pool

To avoid the overhead from thread creation each time a parallel region is created, libGOMP manages a thread pool. At the end of a parallel region the thread is not removed, but stored in a thread pool for later reuse. The current implementation of the thread pool has one limitation: Threads from nested teams are not store. For nested parallel regions, the threads of the inner parallel regions are terminated at the end of the inner parallel region rather than stored in the thread pool. The main reason for discarding inner-nested threads is that additional locking would be required when the thread pool is accessed. But locking is not necessary when there is only one nesting level stored in the thread pool, as the whole thread pool can be managed by the initial master thread.
The fact that nested threads are not cached is expected to decrease the performance for strongly nested programs. On the other hand, not caching nested threads will increase performance for non-nested programs, as there is no additional locking overhead. But whether the locking overhead is really significant and if the overhead of locking exceeds the performance gains provided by the caching of nested threads, must be further investigated.

3 Design

This chapter describes the potential benefits and design considerations of introducing dynamic threading to the OpenMP standard. Furthermore, the consequences of the necessary changes to the OpenMP standard are discussed in detail.

3.1 Performance Characteristics of Nested Parallel Regions

Non-nested parallel programs perform well with OpenMP, since single level parallelism can nicely be balanced over the execution units in most case, except for statically scheduled loops. E.g., consider the two main work-sharing constructs, loops and sections:

- Non-nested parallel sections are processed by assigning a worker-thread to each section. When there are more section constructs than worker-threads, the method of scheduling the structured blocks among the threads in the team is implementation defined. But most elaborate implementations, as e.g. libGOMP, will as soon as a thread becomes idle, assign a section to the idle thread, such that there are no idle threads as long as there are sections left to be executed.

- For a non-nested parallel work-sharing loop, the iterations can be distributed statically or dynamically over the execution units. A static distribution works best, if every loop iteration takes about the same amount of time. Static scheduling incurs least runtime overhead, since locks are not required to distribute loop iterations.
  Having different execution times for different loop iterations, the iterations should be distribute by using the schedule kinds `dynamic` or `guided`. Scheduling with schedule kind `dynamic` or `guided` introduces a runtime overhead but is more efficient in terms of irregular iterations, as each thread will fetch a new chunk of iterations as soon as the thread gets idle.

The distribution of work among execution units is more complex for nested parallel regions. First of all there are problems with the standard itself for nested parallelism as e.g., that there is no means to get a global thread id, which makes it hard for the programmer to analyze which thread has executed which region. Furthermore, the support for `threadprivate` data is limited in a nested region. The mentioned limitations and others are further described in section two of [8].

However, the issue this thesis focuses on is, that the fork-join execution model of OpenMP requires a region to have a fixed-size team while being executed. Fixed-size teams imply that when having two parallel teams and one of the teams finishes, the threads of the finished team cannot join the other team, as the size of the active team has to remain constant. As a consequence, the resources of the finished parallel region cannot be made available to the active parallel region.

To illustrate the problem in more detail, consider the following example: Assume an 8 core machine and having two parallel sections with each a parallel work-sharing loop in the section, as shown in Figure 3.1. Each inner parallel region is assigned a fixed number of execution units (e.g. 4). Assume that the second loop performs a million iterations, while the first parallel loop performs a single iteration. The first section will finish prior to the second and wait for the second section at the implicit barrier at the end of the outer work-sharing construct. Meanwhile, 4 execution units remain idle. The second section, which potentially profits from more worker-threads has no chance to acquire the idle resources. This agnosticism between different teams reduces potential performance gains significantly.

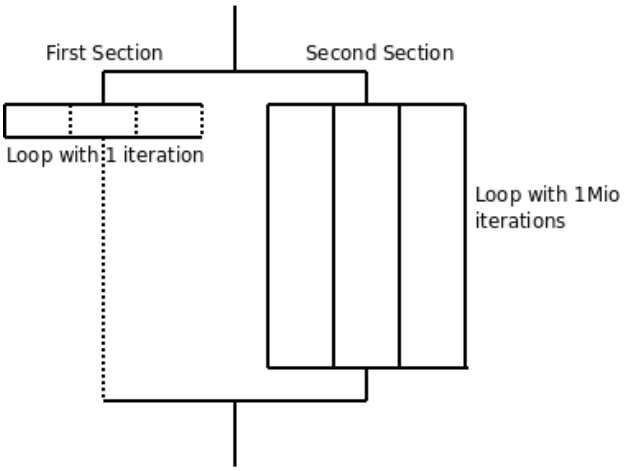

Figure 3.1: A lazy and a busy parallel work-sharing loop on two separate parallel sections.

3.2 Dynamic Threading

To address the idleness of the mentioned resources, *dynamic threading* for OpenMP is proposed. Dynamic threading enables the joining of idle threads to active parallel work-sharing loops at runtime.

Dynamic threading focuses on loops only. The reasons why only loops are considered are, because loops are the constructs in which most time is spent in and because loops are the only construct that allows other threads to easily join the execution, as (at least in data-parallel loops) each iteration can be executed individually. To make threads shareable between different teams, a new loop scheduling kind is introduced - `adaptive`. The `adaptive` scheduling kind behaves similar to the `guided` scheduling kind, but differs in two important aspects:

Firstly, at the start of an adaptive loop, the first thread that enters the adaptive work-sharing construct inserts a function pointer to the outlined loop body into a global function pool. The implementation of the function pool is further described in Section 4.2.1. In addition, the function data and the team state of the thread that is intended to execute the new parallel adaptive loop is provided, such that new threads have all the information that is needed to join the iterations of the adaptive work-sharing loop.

Second, when an adaptive work-sharing construct is finished the function pointer to the finished work-sharing construct is removed from the global function pool. Removing finished work-sharing regions is important to be implemented correctly to avoid deadlocks, because otherwise cyclic helping among threads would become possible and could therefore lead to deadlocks.
After the finishing thread has removed the adaptive work-sharing region from the function pool, the current work-sharing construct can be terminated and thereafter the finishing thread can check if there are other active, adaptive loops. If there are, the current thread assists the execution of the corresponding loop iterations as illustrated in Figure 3.2. The address of the outlined loop body can be found in the global function pool. The helper-thread will execute iterations of the joined loop until there are no iterations left to be executed.

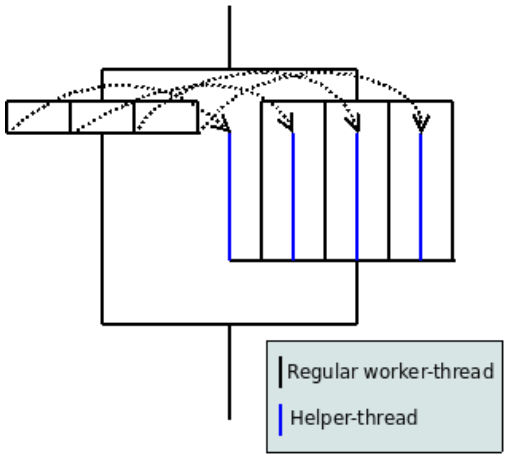

Figure 3.2: Two irregular nested parallel adaptive work-sharing loops. The threads of the first adaptive work-sharing loop join the second active adaptive work-sharing loop.

When a helper-thread finished an adaptive region, the helper-thread will join back to the original team as shown in Figure 3.3. If all team members are finished, the threads will terminate, along with the whole team, except for the master thread.

3.3 Design Decisions

3.3.1 Loops only

To keep the changes to the compiler and runtime library at a reasonable level, dynamic threading was implemented for loops only. An advantage of sticking to loops only is, that the OpenMP

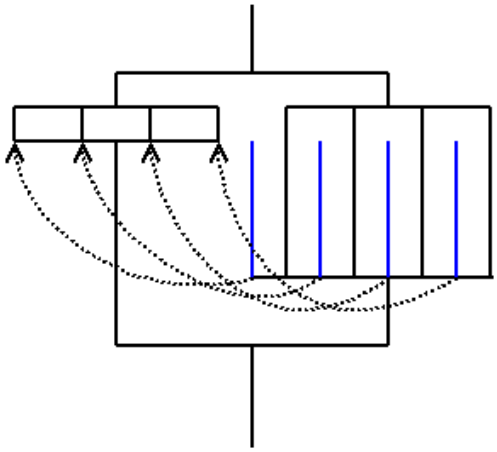

Figure 3.3: Two irregular nested parallel adaptive work-sharing loops. The threads that were helping in the second adaptive work-sharing loop return back to the original team.

syntax can simply be extend by a new scheduling clause to support the improvements, while still profiting from a loop optimization, which have a potentially big impact on the overall execution time. Furthermore, dynamic threading for loops only, does not cause unexpected addition or removal of threads to non-adaptive regions, as the effects of the `adaptive` scheduling clause come only to appearance in adaptive loops.

3.3.1.1 Alternative - Threads Help after Team End

An alternative thread-scheduling heuristic (which is not implemented in our prototype) is to e.g., use all threads that get idle to help active adaptive work-sharing regions. Hence threads do not join another team when the *work-sharing* region is finished, but when the threads reach the end of a *parallel region*. In particular a team of a parallel region must process all work-sharing constructs before the worker-threads are allowed to join a different team.

An example where the alternative approach could be positive is shown in Figure 3.4. Assume having two parallel sections, where the first section has one big parallel adaptive work-sharing region and the second section has a small adaptive work-sharing region followed by a big non-adaptive region. The worker-threads of the second section will continue executing in the same team after the small adaptive loop is finished until the whole parallel region is finished, before helping any other teams. But by then the first section would also be finished and no helping would be necessary.
On the other hand, with the approach that is proposed in this work, after having executed the small adaptive loop of the second section, the threads would instantly join the big adaptive loop in the first section, although there is a lot of work in the original parallel region of the thread.

The disadvantage of the alternative thread scheduling variant is, that the implementation requires additional changes to the code of GCC, as not only loop constructs need to be modified but also the parallel region constructs. To control, which parallel regions offer helper-threads, an additional mechanism is needed to identify the parallel regions.
Furthermore, the alternative approach could cause problems when threads that are helping another

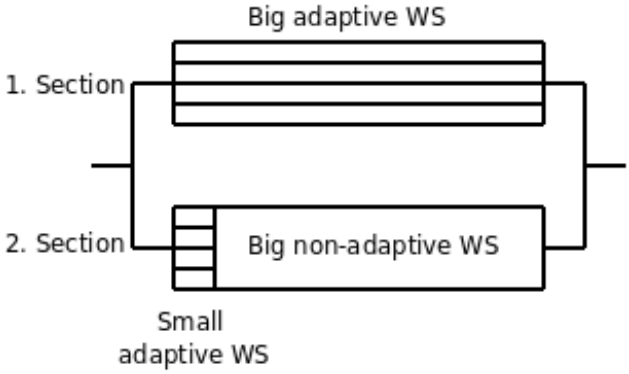

Figure 3.4: Two nested parallel, partly adaptive work-sharing loops.

First, consider the case where threads of an adaptive region help another active adaptive region as soon as the threads reach the end of the *work-sharing region*: The threads of the small adaptive work-sharing region in the second section will help the big adaptive work-sharing region of the first section as soon soon as the small adaptive work-sharing region is finished. Then, the threads of the second section will execute the big adaptive region until the end, return back to the original team and finish the execution of the remaining non-adaptive parallel region, without any help of the threads of the other team.

Now, consider the case where threads help another active adaptive region when the threads reach the end of the *parallel region*: The threads of the upper and the lower parallel region will both execute the parallel regions to the end, to then check if there are other adaptive regions left to be executed.

Assuming both parallel regions take the same time to be executed sequentially, more time is needed to execute the whole parallel region when using the first approach than when using the second approach. Using the first approach, the threads of the second section will, when helping, decrease the execution time of the team of the first section, but on the other hand increase the time for the team of the second section. As both parallel regions have the same length, the first approach will increase the total execution time, as the best strategy would have been not to help at all.

adaptive team, are needed again by the master thread of the original team. E.g. when there are two parallel regions in sequence, as shown in Figure 3.5, the second parallel region will not have any threads left when the first parallel region has offered all the helper-threads. The master thread of the second parallel region would either have to recreate new threads or wait for the threads of the old team, which could again be difficult, because the threads would not belong to the original team anymore, as the threads needed to leave the team in order to get idle. If threads were recreated, more threads than physical cores could be created, which leads to unnecessary thread scheduling which in turn decreases the performance.

Moreover, the alternative approach would also introduce problems with `threadprivate` variables, as when recreating new threads, the `threadprivate` variables could not be maintained. The reason why `threadprivate` variables cannot be maintained is, that `threadprivate` variables are attached to a thread. When threads are sent to help elsewhere, and new threads must be recreated, the values of the `threadprivate` variables of the previous threads are not available anymore.

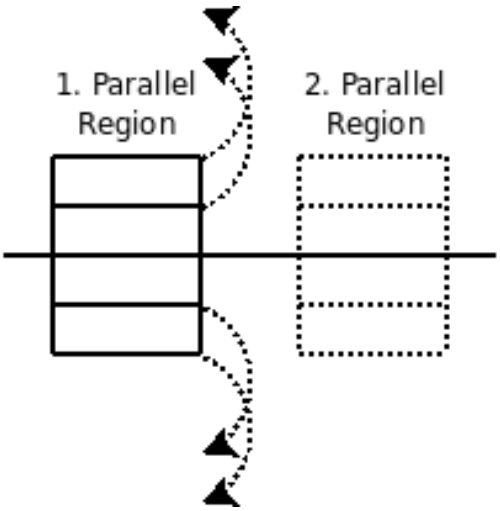

Figure 3.5: Helper-threads leave the first parallel region and do not return for the beginning of the second parallel region.

3.3.1.2 Alternative - Master thread continues execution

Another alternative to the proposed approach is, having the master thread continue in the original team, in order to detect new adaptive regions. That way, when another adaptive region terminates, a new region will already be in the function pool and can therefore be joined.
Keeping the master thread in the original team would on the other hand also reduce the helper team by one thread and maybe cause one more idle thread when there is no work left in the original team.
Furthermore, having the master thread continuing in the original team would also lead to problems when the threads of one team are working on two work-sharing regions, as the OpenMP standard and the implementation are tuned towards having only one work-sharing region per team and therefore would result in a big restructuring of the GOMP library.

3.3.2 Iteration Scheduler

As the proposed scheduling variant only handles how threads are passed between different teams, there still had to be decided how the loop iterations are divided between the different team members. Therefore, existing scheduling variants as e.g. `static`, `dynamic` or `guided` could be used, or a new scheduling variant could have been introduced. But as the focus of this work lies on the proper thread distribution between the different teams rather than on the scheduling in the team, an existing scheduling variant was used. From the existing scheduling variants only a dynamic scheduling variant could be used, as `static` scheduling distributes loop iterations at loop entrance and would therefore not allow any new threads to join the loop at runtime. From the dynamic scheduling variants (`dynamic` and `guided`), `guided` has been chosen, because the `guided` scheduling variant has less runtime overhead than `dynamic` and is therefore a good trade-off between static and dynamic scheduling.

3.4 Comparison Between Dynamic Threading and the Task Construct

The `task` construct has been introduced in version 3.0 of the OpenMP standard[6] and provides an alternative approach to deal with irregular parallelism. A thread that encounters a `task` construct creates a task out of the following structured block. The data-sharing attributes of the variables used within the task may be regulated using data-sharing clauses. Tasks are work units that can be executed immediately by the encountering thread or can be deferred to be executed by a thread of the current team at a later time. Furthermore, there are also task specific synchronization constructs, as e.g., `taskwait`, that ensure that the work of a task is finished at a certain point in the code. The reader is encouraged to have a look at the OpenMP specification for more detailed informations about tasks.

An introduction to OpenMP 3.0 and the task construct has been presented in [17] and an excerpt thereof is presented in Listing 3.1. The listing shows an example how the Fibonacci numbers can be computed using tasks. To compute the Fibonacci number for a number n, the previous two Fibonacci numbers are computed by individual tasks, which both must be finished before the `taskwait` construct can continue.

Listing 3.1: Computation of Fibonacci numbers using tasks.

```
int fib ( int n )
{
    int x,y;
    if ( n < 2 ) return n;
    #pragma omp task shared(x)
    x = fib(n-1);
    #pragma omp task shared(y)
    y = fib(n-2);
    #pragma omp taskwait
    return x+y;
}
```

Tasks and dynamic threading provide both a way to deal with irregular parallelism. However, the irregularity that can be handled by tasks is limited to one level of parallelism, as tasks can only be executed by the threads of the current team. Dynamic threading does not impose any limitation to the level of parallelism, as the workloads of a loop can be executed by the threads of any team with an adaptive region.
Furthermore, tasks provide a more construct independent approach to identify work units, as nearly any code can be marked as task that can be executed by individual threads, whereas with dynamic threading, only loops can profit from helper-threads.

3.5 Changes to the OpenMP Standard

The `adaptive` scheduling variant introduces inconsistencies with the OpenMP specification 2.5 [5].

Most importantly, the fork-join execution model of OpenMP does not hold for adaptive work-sharing regions. According to the fork-join execution model, every new thread is forked from his parent thread, helping the team of the parent thread and when the thread reaches the end of the parallel region, the thread is joined with his parent again. While the manner of forking and joining is maintained, the threads are enabled to engage in the work of other teams before joining back with the parent thread. Changing the execution model is essential for the proposed scheduling variant and therefore cannot conform to the OpenMP specification. But the proposed scheduling variant does not have any influence on the other scheduling variants and preserves the fork-join execution model with the other scheduling variants.

Furthermore, the OpenMP standard specifies that the number of threads in a team must stay constant while a parallel region is executed. The proposed scheduling variant cannot conform to the specification, as helper-threads may join other teams.

Additionally, work-sharing loops must support the `adaptive` schedule kind. Setting the `adaptive` schedule can be done statically by setting the loop schedule clause to `adaptive`, or dynamically by setting the `OMP_SCHEDULE` environment variable to `'adaptive'`. Additionally, our implementation also allows the setting of the chunk size.

As described above, the implementation of dynamic threading breaks with the OpenMP specification. But on the other hand, the OpenMP syntax is not changed. An original OpenMP program can be translated with our modified version of the GCC without the need to make any changes to the code.

4 Implementation

This chapter describes the implementation of dynamic threading in more detail. The implementation touches three main aspects of the existing OpenMP implementation: Firstly, the front end, which parses OpenMP code to GENERIC IR. Secondly, the middle-end, which lowers OpenMP constructs to calls to the runtime library. Finally, the runtime library, which implements the OpenMP API.

4.1 Parser and IR

In order to implement the new scheduling kind, minor changes to the parser and IR were made. In particular, the new schedule kind is added to the struct of existing schedule kinds and the C, C++ and Fortran parsers are modified to accept the new scheduling kind. Furthermore, the OpenMP expansion pass has to ensure that the right functions in the libGOMP runtime library are called.

4.2 Extended OpenMP Runtime Library

The following section describes the changes to the libGOMP in more detail.

gomp_loop_adaptive_start is called before entering an adaptively scheduled loop.
`gomp_loop_adaptive_start` passes the entry function pointer for a loop body to the runtime library. The function pointer is necessary since if a new helper-thread wants to join the execution of another active work-sharing region, the threads must know the address of the outlined loop body. Furthermore, `gomp_loop_adaptive_start` is also responsible for the creation and initialization of the `gomp_work_share` struct. The initialization of the `gomp_work_share` struct involves the setting of the `schedule` variable to indicate the schedule kind, the `chunk_size` that stores the size of an iteration chunk, the `next` and `end` variables containing the lower and upper bound of the iteration variable, and the `incr` variable which contains the value by which the iteration variable is increased after every iteration.

gomp_loop_ordered_adaptive_start is similar to `gomp_loop_adaptive_start`, but additionally initializes the structures needed for the ordered clause, which is described in Section 2.1. The initialization involves calling `gomp_ordered_first`, which associates an order among the threads in the team such that the `ordered` regions are executed in sequential order.

gomp_loop_adaptive_next is responsible for the retrieval of the next chunk of iterations to be executed.

gomp_loop_ordered_adaptive_next also returns the next chunk of iterations to be executed, but additionally ensures that only the thread whose turn it is, is run.

gomp_parallel_loop_adaptive_start is used to store the entry function pointer, the data pointer and the current `gomp_team_state` to the function pool, in case of a combined parallel loop with `adaptive` or `runtime` schedule. This information is mandatory for helper threads to join an active adaptive region.

gomp_loop_end is modified such that the first thread, that reaches the end of an adaptive work-share construct, unregisters the processed adaptive work-sharing construct from the global function pool. Thereafter, the thread calling `gomp_loop_end` will unregister from, or if the calling thread is the last thread that leaves the work-sharing region, terminate the `gomp_work_share` struct. After having left the work-sharing region, the current thread checks for other active adaptive loops in the function pool and joins the corresponding loop.

gomp_loop_end_nowait does basically the same as `gomp_loop_end`, but `gomp_loop_end_nowait` implements the nowait semantics. That is, there is no implicit barrier at the end of the work-sharing construct.

4.2.1 The Function Pool

In addition to the changes of the libGOMP, a global function pool is added. The pool is implemented as a linked list with a pointer to the first and the last element. Thread safe entries to and removals from the function pool are guaranteed by a lock.
Each function pool entry consists of a function pointer, a pointer to the function's data, a team state, to store the previous team and work-sharing region, and a pointer to the next element in the pool. The corresponding function pool structs are shown in Listing A.4 of the Appendix. Furthermore, to manipulate the function pool three new functions are provided:

gomp_adaptive_pool_insert inserts a new entry at the end of the function pool.

gomp_adaptive_pool_remove removes the entry that corresponds to the current adaptive loop from the function pool.

gomp_adaptive_pool_fetch_execute checks for an entry in the function pool and, in case joins the adaptive region that has been found. To join another adaptive work-sharing region, the current thread calls `work_share_join` which is responsible for reinitializing the thread and calling the active adaptive work-sharing region. The reinitialization of the thread involves resetting the team state, function and data pointer along with a modification to the size of the team and retrieval of the work-sharing region to be executed. Furthermore, the team barriers must be set according to the new number of threads in the team and the ordered structs have also to be resized.

To summarize, a thread that joins another adaptive region must perform the following steps: First, the original adaptive work-sharing region must be executed. After having processed the original region, the corresponding function pointer must be removed from the function pool, to avoid other threads from joining the processed work-sharing region. Then, the current thread terminates the processed work-sharing construct. Hence, the thread is able to join another work-sharing construct. Finally, the thread checks the function pool for workloads. If the function pool is non empty, the thread initializes the corresponding data structures (`gomp_thread`, `gomp_work_share` and `gomp_team`) and joins the active adaptive region.

4.3 Middle End - Loop Body Outlining

In addition to the changes to the front and middle end of the GNU Compiler Collection loop body outlining must be implemented for a special case: Non-combined work-sharing constructs (See Section 2.1). The main reason for outlining loop bodies is, that the runtime library needs an entry point address of the adaptive loop for helper threads. To pass the entry point address of an adaptive region to the `gomp_loop_adaptive_start` function the body of adaptive loops must be outlined into a separate function. As `runtime` loops, which are at runtime assigned to use the `adaptive` schedule kind, should also be executable, the body of `runtime` loops had to be outlined.

Loop body outlining requires modification in the gimplification pass in `gimplify.c`. In particular, variables that are used in the loop body must be propagated correctly to the OpenMP lowering and expansion passes. For variables in parallel regions, that do not have the data-sharing attribute specified by any OpenMP data-sharing clause, the standard defines an implicit data-sharing clause. The implicit data-sharing clauses for variables in parallel regions are provided by the gimplification pass. But with the outlining of adaptive loops, implicit data-sharing clauses have also to be specified for variables within outlined loops, as the variables of the region before the loop have to be passed to the loop body function. The implementation of the code that provides implicit data-sharing clauses for adaptive loops was mostly reused from the code that provides the implicit data-sharing clauses for parallel regions. Therefore, after the gimplification pass, every variable within a parallel region or an adaptive region has a data-sharing attribute assigned.

Most of the modification in the middle end are implemented in `omp-low.c`, which includes two passes that are responsible for the generation of OpenMP code. A lowering pass and an expansion pass.

The lowering pass consists of two steps.
The first step scans for adaptive loops and creates a new data-sharing struct, which is similar to the data-sharing struct used with parallel regions, mentioned in Section 2.2.1. The data-sharing struct facilitates the passing of all non-global loop body variables from the parallel region to the loop body function.
The second step implements the actual lowering of the adaptive loop region, which consists of the gimplification of variables in special clauses as e.g., `reduction` or `lastprivate` and the generation

of code to pass all the involved variables from the outer-loop region to the outlined loop body function. The passing of the variables between the different scopes is implemented by generating code to fill the data-sharing struct. Thereafter, the generated code can be composed to a list of statements that is used to generate the new function.

The lowering pass is followed by other optimization passes, that e.g., create the CFG. The last OpenMP related pass is the expansion pass:

The expansion pass is executed for every function and first of all scans for OpenMP regions. Every OpenMP region is expanded correspondingly:

- Expanding a parallel region comprises the outlining of the parallel region into a new function and calling the `gomp_parallel_start` and `gomp_parallel_end` runtime library function to start and end a team of threads (See Listing 2.4 and 2.5). If the parallel region is a combined parallel region, the expansion will use a call to `gomp_parallel_loop_*_start` instead of a call to `gomp_parallel_start`, such that the loop can be initialized along with the parallel region and to minimize the locking overhead.
- To expand a loop, the abstract loop statement has to be expanded into a loop initialization part, a loop body part, a chunk retrieval part and a cleanup part.
  - The loop initialization part initializes the loop body variables and if necessary creates a call to the `gomp_loop_*_start` function, which is responsible for initializing the loop work-sharing struct (See Section 4.2). The call to `gomp_loop_*_start` is only necessary when the current region is not a combined parallel region, as for combined parallel regions the call to `gomp_parallel_loop_*_start` will have already initialized the loop work-sharing struct.
  - The loop body part consists of the code that is executed within the loop along with the incrementation of the loop iteration variable and the code to check if the end of the current chunk of iterations has already been reached. A jump instruction to the chunk retrieval part is executed when the loop body part has no more iterations left in the current chunk.
  - The chunk retrieval part will call the `gomp_loop_*_next` function to retrieve another chunk of iterations and jump back to the loop body part. If there are no more chunks of iterations available, a jump instruction to the loop cleanup part is executed.
  - The loop cleanup part creates the call to the `gomp_loop_end` or `gomp_loop_end_nowait` function, which are responsible for terminating the loop work-sharing struct.

  Finally, the loop expansion will reconnect the edges of the basic blocks of the loop.
- The expansion of an adaptive loop happens in two expansion passes:
  The first time the expansion pass is executed for a non-combined adaptive loop, the adaptive loop is outlined into a separate function as described for the expansion of parallel regions. However, there are certain differences as e.g., that the `gomp_parallel_end` function is not

called and instead of the `gomp_parallel_start` function, the `gomp_loop_adaptive_start` function is called. For combined parallel adaptive loops, the expansion is similar to the expansion of parallel regions, except that the `gomp_parallel_loop_adaptive_start` function is called.

The second time the expansion pass is executed, the loop of the outlined loop body function is expanded, as described for the expansion of loops, except for a subtle difference: The loop initialization has to consider variables that must be initialized specially as e.g., `reduction` variables.

5 Performance Evaluation

This chapter describes how the performance of the proposed implementation has been evaluated. Firstly, the experimental settings are specified. Then, the benchmarks are described and the results are discussed.

5.1 Experimental Settings

The performance evaluation was conducted on an Intel Xeon (E5450) quad-core with 3GHz clock speed and hyper threading. The system has a bus speed of 1333 MHz, 8GB DDR2 RAM, 6MB shared L2 cache per core pair, giving a total of 12MB L2 cache, as well as a total of 256KB L1 cache. The system is running Ubuntu 8.04.1 (Hardy Heron) with kernel 2.6.24-19.

5.2 Benchmarks

To evaluate the proposed implementation, three synthetic benchmarks have been conducted. The execution times for the benchmarks are averaged over 100 executions and each benchmark is executed with all scheduling kinds (`static`, `dynamic`, `guided` and `adaptive`).
The workload of the synthetic benchmarks is to count the number of primes to a given upper boundary. The workload is chosen arbitrarily and scales well with additional threads. Similar performance results for similar workloads are expected.

irreg-prime-good has two parallel sections with each a parallel loop that computes the number of primes up to a certain number. `irreg-prime-good` was executed multiple times, whereas the workload of the two parallel sections was changed. The variation of the work-sharing sizes is supposed to demonstrate how different levels of irregularity influence the behavior of dynamic threading. A sketch of `irreg-prime-good` can be seen in Figure 5.1.
`irreg-prime-good` is expected to perform better with the `adaptive` scheduling kind than with the other scheduling kinds, as the threads of work-sharing region 1 are able to join work-sharing region 2 and therewith decrease the execution time of work-sharing region 2.

irreg-prime-bad has two parallel sections. The second section calculates in a parallel work-sharing loop the number of primes up to a certain number (N). The first section first cal-

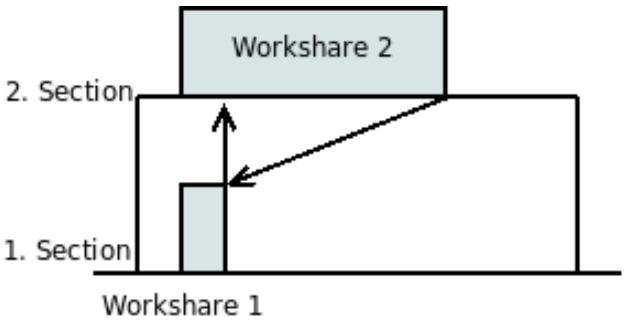

Figure 5.1: An irregular, nested parallel program that performs well. The threads of the adaptive work-sharing region 1 will upon termination of work-sharing region 1 join the big adaptive work-sharing region 2 of the second section. After the big work-sharing region 2 has finished, the threads that originally belonged to work-sharing region 1 will return to the original team and in case of a worker-thread terminate or in case of the master thread continue to the barrier at the end of the non-nested parallel region. The threads of the team that started work-sharing region 2 will also terminate the work-sharing and parallel region and the master thread will proceed to the barrier.

culates the number of primes to a certain number (M) before counting the primes till N. `irreg-prime-bad` was executed multiple times, each run with different numbers for N and M. A sketch of `irreg-prime-bad` where M is smaller than N is illustrated in Figure 5.2.
`irreg-prime-bad` shows a negative scenario of the current implementation of dynamic threading. The threads of work-sharing loop 1 in the first section join work-sharing loop 2 of the second section as soon as the threads are finished with work-sharing loop 1. Having jointly executed work-sharing loop 2 of the second section, the threads of the first section will return to the original team, whereas the threads of the second section will terminate. The threads of the second section terminate because at the time of the termination of work-sharing loop 2, the first section has not started work-sharing loop 3 yet. Therefore, the threads in the first section must execute the remaining work-sharing loop 3 without any help of other helper-threads.

simple-para-loop is a single parallel work-sharing loop that counts the prime numbers up to a certain number. `simple-para-loop` has been executed multiple times, each time with different number of iterations.
`simple-para-loop` is used to reveal the overhead that is introduced by the `adaptive` scheduler.

5.3 Results

This section shows and discusses the results for each of the benchmarks.

irreg-prime-good The results of benchmark `irreg-prime-good` are presented in Table 5.1 and in normalized form in Figure 5.3. The relations of Figure 5.3 are relative to a work-sharing loop with

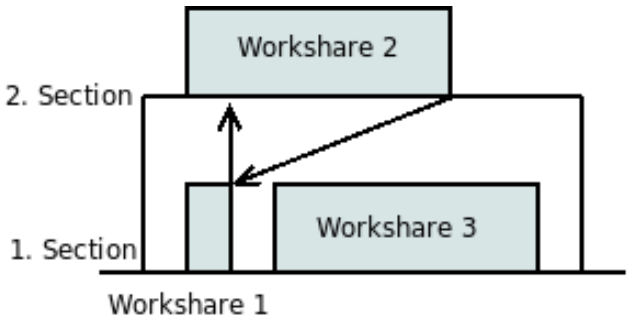

Figure 5.2: An irregular, nested parallel program that performs bad. The threads of the adaptive work-sharing region 1 will upon termination of the work-sharing region 1 join the big adaptive work-sharing region 2 of the second section. After the big work-sharing region 2 has finished, the threads that originally belonged to the small work-sharing region 1 will return, and the threads of the big work-sharing region 2 will terminate. The threads of work-sharing region 2 terminate as by the time the big work-sharing region 2 is finished, the big work-sharing region 3 has not been reached by any thread yet and can therefore not be joined. Therefore, the threads of work-sharing region 1, that returned from work-sharing region 2, must execute work-sharing region 3 without any help from other helper-threads.

10000 iterations and the baseline for the normalization is the `static` scheduling kind.
The results show, that as soon as the workloads of the sections get more irregular, the `adaptive` schedule kind performs better than the other scheduling kinds. The mean execution times of `adaptive` decrease up to nearly the half of the other scheduling kinds, as the number of threads that are executing the bigger parallel region is doubled.
Furthermore, the results show also, that if the workloads get smaller, dynamic threading cannot perform quite as well as with bigger workloads. A reason for the decreased performance gain with smaller sections is that the overhead to switch the threads from one team to another, increases in relation to the actual work that is done within the work-sharing loops. That is, while the team-switch took only a fraction of the total time with bigger workloads, with smaller workloads the team-switch needs more of the total time and therefore has a bigger influence on the result.

irreg-prime-bad The results of benchmark `irreg-prime-bad` are presented in Table 5.2 and in normalized form in Figure 5.4. The relations of Figure 5.4 are relative to a work-sharing loop with 10000 iterations and the baseline for the normalization is the `static` scheduling kind. That is (e.g. in Figure 5.2) work-sharing region 1 has 10000 iterations while work-sharing region 2 and 3 have the corresponding percentage of iterations.
The results show, that the performance with the `adaptive` scheduling kind is worse when the workload of work-sharing regions 2 and 3 is bigger than the workload of work-sharing region 1. The reason for the bad performance is that the threads of work-sharing region 2 are not able to join work-sharing region 3, as work-sharing region 3 has not yet started and is therefore not registered in the function pool. Therefore, the threads of the first section must execute work-sharing region 1, help work-sharing region 2 and finally also execute work-sharing region 3. Therefore, the overall execution time increases by 50% as compared to the execution time of the other scheduling kinds.

What currently cannot be explained is, that when work-sharing region 2 and 3 are smaller than work-sharing region 1, the `adaptive` scheduling kind performs worse than the other scheduling kinds. The expected results for a smaller work-sharing region 2 and 3 would be that adaptive scheduling outperforms the other scheduling variants, since the threads of work-sharing region 2 will finish prior to the threads of work-sharing region 1 and can join the big work-sharing region 1, which can profit from additional worker-threads.

simple-para-loop The results of benchmark `simple-para-loop` are presented in Table 5.3 and in normalized form in Figure 5.5.
The results shows that the `adaptive` scheduling kind does not introduce a significant overhead, as the only additional overhead results from the insertion of an entry to the function pool at the beginning of an adaptive region and a removal of the current adaptive loop from the function pool at the end of an adaptive loop.
However, `simple-para-loop` does not show the overhead that is produced when joining another adaptive region. The reason that no benchmark is provided that shows the overhead of joining an adaptive region is, that to measure the overhead, another adaptive region would have to be joined and therefore executed, which would again influence the mesured time.

Section1	Section2	Relation[$\frac{Section1}{Section2}$]	static[s]	dynamic[s]	guided[s]	adaptive[s]
10	10000	0.1%	0.0646930	0.0652820	0.0599220	0.0580930
100	10000	1%	0.0632720	0.0591000	0.0560970	0.0625330
1000	10000	10%	0.0578180	0.0593340	0.0541310	0.0620140
5000	10000	50%	0.0833390	0.0685980	0.0616520	0.0801190
10000	10000	100%	0.0952370	0.0763800	0.0785120	0.0818390
25000	10000	250%	0.2800340	0.2902280	0.2874280	0.2228470
50000	10000	500%	1.0923790	1.0813070	1.0784990	0.6560480
75000	10000	750%	2.3964080	2.3839440	2.3832460	1.3539420
100000	10000	1000%	4.2055000	4.2097020	4.2088990	2.3983830

Table 5.1: Mean execution times for irreg-prime-good.

Section1	Section2&3	Relation[$\frac{Section1}{Section2\&3}$]	static[s]	dynamic[s]	guided[s]	adaptive[s]
10	10000	0.1%	0.054189	0.060894	0.044311	0.087479
100	10000	1%	0.047596	0.119633	0.066250	0.057026
1000	10000	10%	0.100971	0.059850	0.052675	0.044190
5000	10000	50%	0.109344	0.068493	0.083626	0.064320
10000	10000	100%	0.110105	0.219843	0.109445	0.128804
25000	10000	250%	0.354831	0.374943	0.376111	0.627750
50000	10000	500%	1.262460	1.115265	1.162816	1.751019
75000	10000	750%	2.489580	2.465208	2.460914	3.725959
100000	10000	1000%	4.320140	4.414614	4.367499	6.596685

Table 5.2: Mean execution times for irreg-prime-bad.

Iterations	static[s]	dynamic[s]	guided[s]	adaptive[s]
1	0.000028	0.000026	0.000026	0.000029
10	0.000026	0.000026	0.000026	0.000026
100	0.000048	0.000047	0.000048	0.000050
1000	0.000661	0.000816	0.000781	0.000630
5000	0.006149	0.006567	0.007212	0.005710
10000	0.023595	0.023605	0.022705	0.022705
25000	0.133706	0.132914	0.132936	0.131186
50000	0.542525	0.529685	0.543449	0.533037
75000	1.190360	1.187521	1.184032	1.187469
100000	2.096478	2.099319	2.096139	2.093886

Table 5.3: Mean execution times for simple-para-loop.

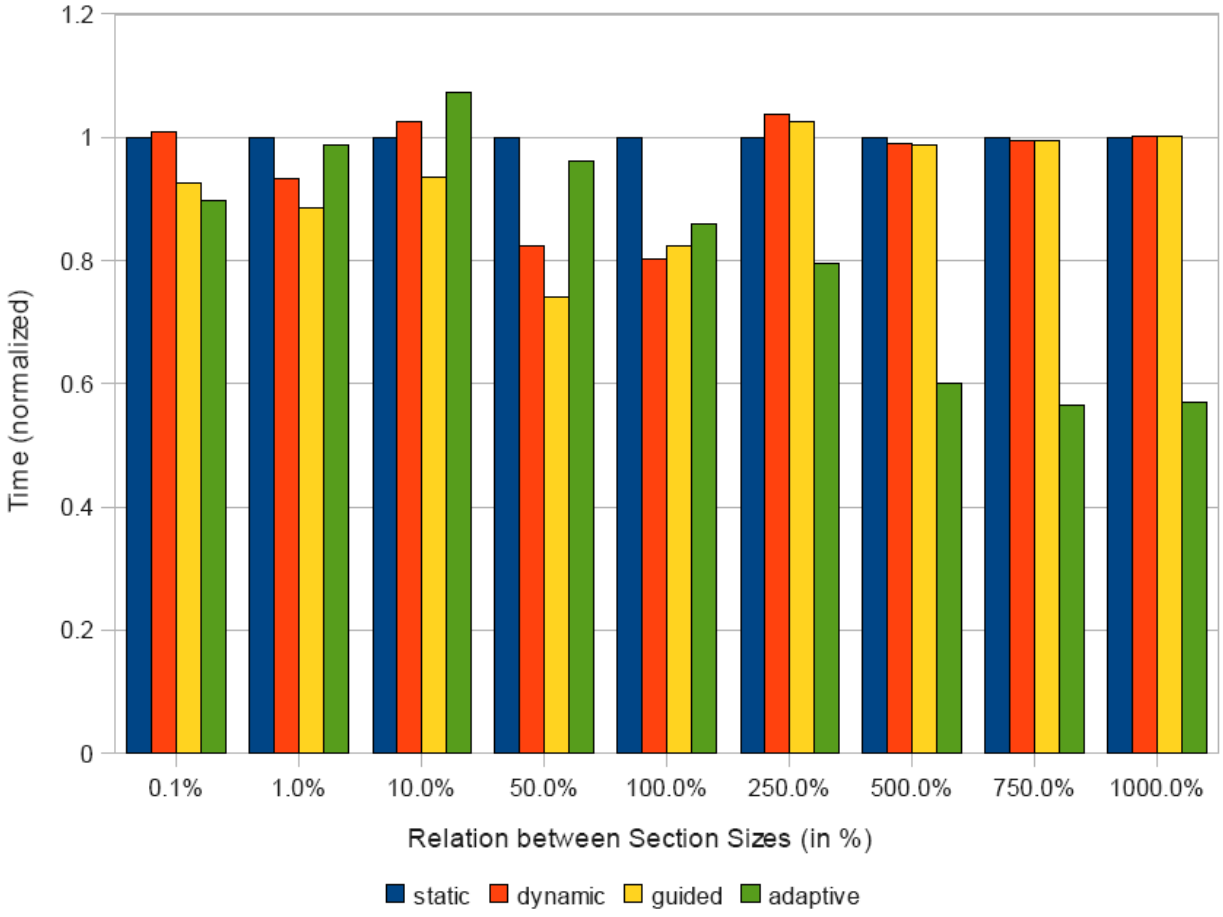

Figure 5.3: Normalized mean execution times for irreg-prime-good.

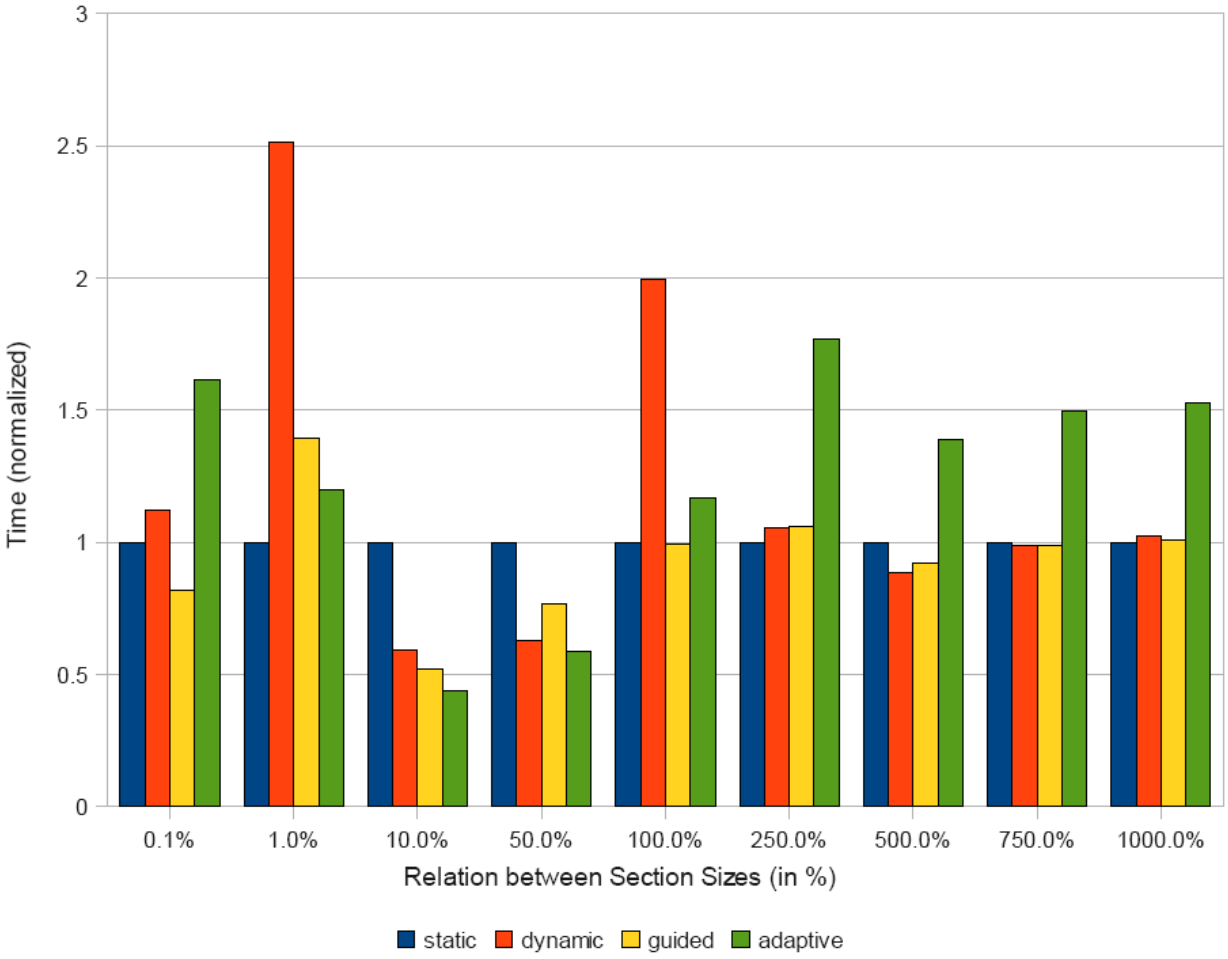

Figure 5.4: Normalized mean execution times for irreg-prime-bad.

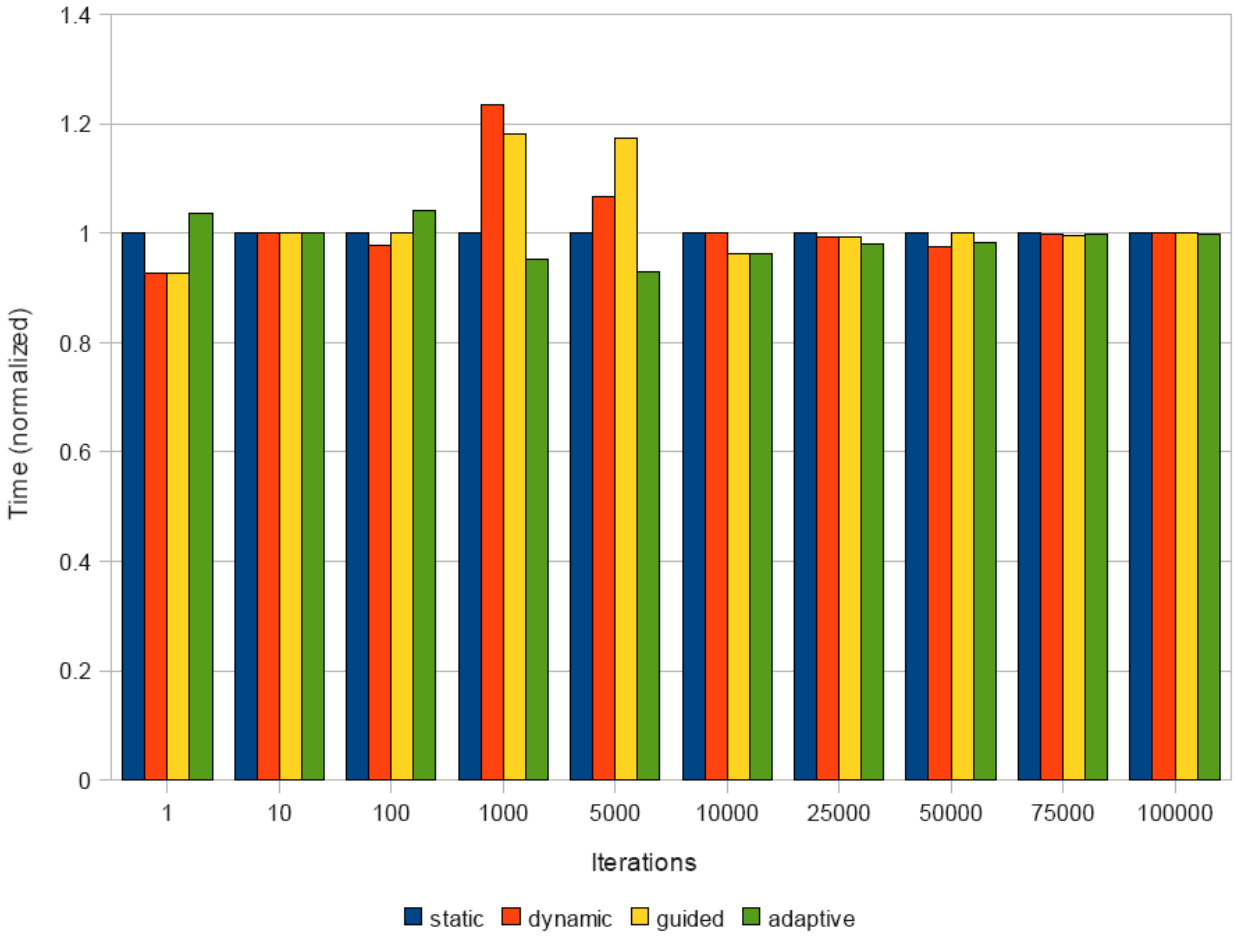

Figure 5.5: Normalized mean execution times for simple-para-loop.

6 Related Work

6.1 Parallel Programming Models

There are numerous parallel programing models and techniques, whereas most are targeted towards a specific problem set or hardware architecture. A good overview over the different parallel programing models can be found in the first chapter of [13].

6.1.1 Manual Parallelization

The most efficient, but most difficult way to parallelize a given program is by manually dividing the sequential program into different threads that can be executed on individual execution units. Manual parallelization gives a much better control over the individual threads, but to the cost of having to deal with low level thread control. E.g., dividing the execution of a loop between different execution units requires the creation of several threads. Each thread is assigned a dedicated part of the loop, which introduces a lot of code, to create, synchronize and join threads, even though the parallelization could be done semi-automatically.

6.1.2 Semi-Automatic Parallelization

A more practical way to parallelize a program is by using semi-automatic parallelization approaches as e.g. OpenMP. The advantage of semi-automatic is, that the user only has to deal with higher level constructs to control the threads. Semi-automatic parallelization gives the application programmer a simpler way to parallelize an existing sequential program, while maintaining enough control to decide how and which code parts should be parallelized and executing how may threads.

6.1.3 Fully Automatic Parallelization

Automatic parallelization or implicit parallelization can be performed at different levels.

The lowest level is instruction pipelining, that is done at hardware level, and allows multiple instructions to be executed in parallel on the processor. For more information about instruction pipelining, the reader is encouraged to have a look at [25].

The next higher level includes vectorization. Vectorization is implemented by the most of the current compilers, including the Intel Compiler and the GCC. Vectorization can be used when an instruction is consecutively executed on multiple operands. The sequence of instructions are transformed into a single instruction multiple data (SIMD) instruction, such that the instruction can be executed on multiple operands at once. SIMD instruction sets, as e.g. the various versions of the Streaming SIMD Extensions (SSE), are available on most current processors.

Finally, the most difficult way of automatic parallelization is, automatically parallelizing a sequential program into a multi-threaded program. The most common way automatic parallelization is implemented by compilers, is by parallelizing loops by inserting OpenMP constructs before parallelizable loops. To find parallelizable loops, dependence and alias analysis is required. Furthermore, even if a loop is parallelizable, there is no certainty whether there is any profit from parallelizing the loop. Additional overhead may be introduced by synchronization and communication between the executing threads.

The current Intel compiler nicely uses the multiple levels of automatic parallelization (vectorization and automatic parallelization) as described in [28]. The authors divide the iterations of loops between different cores, while having each of the cores using vectorization to perform even better.

The current GNU Compiler Collection on the other hand does currently only support automatic vectorization. Preparations in order to implement automatic parallelization have been taken as described in [24].

6.1.4 Distributed Memory Systems

Even though OpenMP performs well on shared memory systems, OpenMP was not designed for distributed memory systems. For distributed memory systems another method to parallelize programs is needed, as there is normally no shared memory available. Even though there exist several methods to parallelize programs for distributed memory systems, the most prominent approach is using the Message Passing Interface (MPI). MPI is an API specification for communication between computers, tuned towards high-performance computing for scalable and portable applications. There exist several implementations of MPI for various languages as described in [2].

6.2 Adaptive Parallelism

An interesting model for adaptive parallelization is presented by Carriero et al. in [14]. The authors proposed the Piranha model which allows distributed processing resources to adaptively join and leave the execution of a given task. With the Piranha model, idle processing resources in a computer network can be used, and relinquished from the work as soon as the processing resources are needed otherwise. Along with the Piranha model the authors also provide three parallel example applications (Monte Carlo simulations, LU decomposition and domain decomposition) which scale well with the proposed approach.

The Piranha model provides, similar to dynamic threading, an approach to distribute workloads adaptively among various execution units. However the authors concentrated on a distributed memory system, whereas dynamic threading is focused on a local setting with multiple execution units and shared memory.

Feedback based Optimizations Recently there have been several proposals for better loop schedulers for OpenMP. E.g., Suleman et al.[26] proposed two approaches which sample loops at runtime. Bandwidth-Aware Threading (BAT) identifies the minimal number of threads to saturate the off-chip bus and Synchronization-Aware Threading (SAT) identifies the optimal number of threads such that the data synchronization overhead does not decrease performance. To evaluate the performance of BAT and SAT the authors used 12 multithreaded applications from different domains. The results show that by combining BAT and SAT the average execution time could be reduced by 17% and the power by 59%.
The proposed approach of Suleman et al. provides, similar to dynamic threading, an approach to adapt the number of threads at runtime. However, SAT and BAT require an additional sampling of the application before the application is run, to predict the right number of threads. This could lead to problems when the workload of the system differs at sampling and execution time, as the predicted number of threads differs depending on the workload. Dynamic threding however does not require any prior sampling but does also not consider the data-synchronisation overhead.

Another feedback based effort which is supposed to make the schedule clause unnecessary was proposed in [9] and also gives a good overview over previous scheduler improvements. Ayguadé et al. propose a general framework that gathers information over loop iterations at runtime to optimize the loop scheduling. The framework determines how balanced the loop iterations are, that is whether each thread has approximately the same amount of work and changes the scheduler based on this balance characteristic. The authors used multiple programs from the SPEComp suite, the NAS OpenMP benchmarks and a computational kernel that calculates the Legendre polynomial to reveal the performance improvments of the proposed framework. The results showed that performance speedups of up to 44% could be achieved without the programmer having to decide upon the scheduling kind to use.
The approach Ayguadé et al. propose is good, when the programmer does not know which scheduling kind should be used, but it will not provide any improvements in case of irregular nested workloads, as teams will still not share threads. However, the proposed framework could be extended by dynamic threading, such that there is no need for a scheduling clause, even with irregular nested workloads.

Curtis-Maury et al. [15] evaluated the performance of OpenMP applications on simultaneous multithreaded (SMT) and multi-core (CMP) processor. The problem the authors criticize is, that running multiple threads with conflicting resource requirements on the same physical SMT processor, limits scalability and degrades performance. To cope with the limited scalability the authors propose a sampling oriented approach that first tries to execute parallel regions with Hyper Threading enabled and once without Hyper Threading and further on uses the method that performed better.

To measure the performance gains of the proposed approach, the authors used the NAS Parallel Benchmarks along with a mesoscale weather prediction model and a matrix pseudospectrum computation code. The results show that an average overall speedup of 3.9% can be achieved.

Hardware Related Optimizations Zhang et al. [29] criticize the bad performance of OpenMP codes, that put threads with different data sets on the same SMT-Node, with Hyper Threading enabled. The authors extended the OpenMP runtime system of the Omni research compiler [3] with a two stage hierarchical scheduler. One stage is used to decide upon the number of cores to use and the second stage to decide whether hyper threading is used. Zhang et al. used seven benchmark applications to evaluate the proposed runtime system extension. The results showed almost 10% improvement in the average performance of the benchmarks compared to the original parallel applications.

Broquedis et al.[11] presented a thread scheduling policy for irregular and massive nested parallelism over hierarchical architectures. The authors propose extensions to the GNU OpenMP runtime system to transparently decide on the right number of threads per SMT processor. The authors' approach is to attach information to groups of threads, called bubbles, to be able the put threads with a higher affinity closer to each other on the processors. Furthermore, the authors use work stealing, which is aware of the underlaying non uniform memory architecture when load balancing is required. Broquedis et al. evaluated their extensions by running a parallel surface reconstruction application with highly irregular divide-and-conquer parallelizm. The results showed that their approach is highly scalable and produced a speedup of up to 15 times on a 16 core machine, compared to sequential execution and a speedup of up to twice compared to a runtime without the extensions.

Hadjidoukas et al. [20] introduce a new version of the OMPi OpenMP Compiler. The OMPi Compiler is enhanced by a lightweight runtime library that supports user-level multithreading. The thread management adaptively distributes the threads over the execution units and puts nested threads on the same execution unit as the master threads, to favor locality. To avoid idleness of execution units the authors provide a hierarchical work stealing mechanism, which exploits the latest developments of shared memory architecture, as multi-core and SMT processors, by assuming hierarchical groups of threads. Furthermore, the authors measured the overheads of the OpenMP constructs within the OMPi compiler compared to compilers as GCC, ICC or the Omni compiler and evaluated the OMPi compiler using PCURE, a hierarchical data clustering program. The overheads of the OMPi compiler showed to be small relative to the other compilers and PCURE performed best using OMPi.

Task Scheduling Strategies Duran et al.[18] analyzed different scheduling strategies for OpenMP 3.0 tasks. The authors provide a breadth-first scheduler, a work-first scheduler and two cut-off policies. The breadth-first scheduler puts every task into a team pool, while having the parent thread continuing. The work-first scheduler on the other hand tries to follow the serial execution path to better exploit data locality. To reduce the overhead of creating tasks, the runtime can

start tasks immediately, which is referred to as cutting off. The proposed cut-off policies are, to immediately execute a task after a certain maximum number of tasks is reached, or when a maximum task recursion level is reached. The authors propose that work-first scheduling performs best, but because of restrictions in OpenMP a breadth-first scheduler is a better default scheduler for an OpenMP runtime library.

Furthermore, Duran et al. [19] also provided an extension to the OpenMP tasking model that allows dependent tasks to be detected at runtime. Dependant tasks allows performance improvements in case of applications where locality or load balancing is essential. The dependencies among tasks are expressed by specifying the input and output direction of the arguments used in a task. The authors used the SparseLU benchmark to demonstrated that the scalability could significantly be increased.

7 Conclusion

This chapter gives a short summary of the extensions to the GNU Compiler Collection. Furthermore, issues that are currently not solved properly in the compiler are discussed. Moreover, optimization to the dynamic threading approach are presented. Finally the work is concluded.

7.1 Summary

Although OpenMP performs well for regular, scientific computations, irregular workloads can lead to inefficient resource utilization, especially when nested parallel regions are used. The main reason for OpenMP's inflexibility to handle irregular workloads efficiently is the fork-join execution model, which does not allow idle threads to join an active parallel region. Hence, theads waiting at an implicit barrier at the end of a work-sharing construct are not allowd to join another active parallel region, which could profit from additional worker-threads

To overcome the limitation of constant-size teams, the GNU Compiler Collection has been extended by introducing a new OpenMP scheduling variant, `adaptive`, that allows threads that reach the end of a specifically marked work-sharing loop to join another team that is currently working on an adaptive work-sharing loop. With this extension the execution time of OpenMP programs can be significantly decreased, depending on the irregularity of the nested work-sharing loops. On the downside, the current implementation also introduces special cases, where the execution time is slightly increased, as shown in test `irreg-prime-bad` in Section 5.3.

To implement dynamic threading, modifications to various parts of GCC and the GNU OpenMP runtime library were made. First, modifications to the front and middle end were made, to handle the proposed new scheduling variant. Furthermore, the runtime library is adapted, such that the new runtime library calls will adaptively distribute the threads among the adaptive regions. Moreover, a function pool had to be created, such that function pointers to adaptive loops could be stored. With the function pool, adaptive regions can be stored upon entering an adaptive region and threads that leave an adaptive region can find other active, adaptive regions, that can still profit from additional worker-threads.

To store function pointers of non-combined adaptive loops into the function pool, the adaptive loop bodies had to be outlined into a function. The outlining required modifications of the gimplifier such that the gimplifier would also generate data-sharing clauses for the loop bodies that must be

outlined into a seperate function. The data-sharing clauses are necessary, because the variables that are used in the loop need a way to be passed from the region outside the loop to the loop body function. Furthermore, also the OpenMP lowering and expansion passes in GCC were modified, such that the data-sharing structs are filled with the variables that are needed in the loop body function and that the right OpenMP functions are called.

7.2 Future Work

This section describes implementation deficiencies of the current implementation and examines ideas that could be implemented to extend the proposed implementation such that nested parallelism performs even better.

7.2.1 Implementation deficiencies

There are still issues that are not yet handled correctly in the compiler.
Firstly, there is currently a bug in the `lower_omp_for_adaptive` function that causes the modified compiler to produce a segmentation fault if a loop boundary variable (e.g. the initial value or the upper bound) is an expression containing operations.

Also, the `reduction`, `critical` and `lastprivate` clauses in non-combined parallel loops are not handled correctly. For non-adaptive loops, variables that are classified with the `reduction`, `critical` or `lastprivate` clause are passed by reference from the scope before the loop to the loop scope. Note that for non-adaptive regions the scope before the loop is the scope before the parallel region. By passing the reference of the variable, that is outside the parallel region (*outer variable*), every thread in the parallel region has the address of the outer variable and is therefore able to atomically update the outer variable.
Reference passing however is more difficult to implement if the loop body is outlined. Because if the loop body is outlined, the scope that is before the loop region is not anymore the scope before the parallel region, but rather the scope of the parallel region. Therefore, not the address of the variable that is outside the *loop region* must be passed, but rather the address of the variable that is outside the *parallel region*. Otherwise the threads within the loop will not atomically update the variable that is in the sequential region, but rather the private variable of each thread. To solve this problem, the address of the variables from before the parallel region must be propagated into the inner loop region.

Furthermore, `ordered` loops are currently not implemented thread safe in the runtime library. To make the `ordered` loops thread safe, every access to the `ordered_release` struct within the `gomp_team` must be done under mutual exclusion. This additional locking is necessary because when a thread joins another team, the size of the `ordered_release` struct of the other team has to be increased. If threads access the `ordered_release` struct while being increased, the work-sharing loop could get executed in the wrong order.

7.2.2 Proposed Extensions

One way to extend the proposed implementation is to not let the threads help when reaching the end of a work-sharing region but rather when the threads get idle, at the end of a parallel region. Having the idle threads helping other adaptive regions, would make sure that threads are not needed anymore by the thread's team. The proposed approach, however, has drawbacks, as described in Section 3.3.1.1.

Another way to extend the compiler is to let the master thread continue in the original team, after the adaptive loop has terminated, instead of joining the helper-threads. The positive effect of the master thread staying in the original team is, that the master thread can find other adaptive work-sharing regions to put them into the function pool. Therefore, threads of an other, finishing team, will not terminate, when no other adaptive loops are available, but rather join the adaptive work-sharing region that has been found by the master thread. However, the proposed approach also leads to problems as discussed in Section 3.3.1.2.

Furthermore, an investigation whether the penalty of locking the thread pool is bigger than the benefit from caching nested threads could be useful to determine whether nested threads should be stored in the thread pool. The fact that nested threads are not cached in the GCC implementation has been described in Section 2.2.2.1.

As soon as the version 4.4 of the GCC, which implements the OpenMP specification 3.0, is released, a port of the proposed extensions to that version could be beneficial. The new OpenMP standard introduces additional constructs, which might turn out to be useful for dynamic threading:

- The behavior of the new environment variable `OMP_WAIT_POLICY` could be modified, to e.g. enable idle threads to join adaptive regions, instead of busy waiting.
- Extensions to the newly introduced `task`s could be made, in order to add adaptive regions as a kind of task instead of putting the adaptive regions into the function pool.

7.3 Lessons Learned

Extending the GNU Compiler Collection to enable the sharing of threads between different teams has shown to be a difficult task, because of various reasons.

The biggest difficulty concerning the implementation was the loop body outlining. The outlining required a significant amount of work in the GCC middle end, which turned out to be even more intricate than expected, as the code base is huge, the documentation poor and bugs mostly show up various compiler passes later.

Another difficulty was to decide on the thread scheduling policy.
On the one hand, when threads join another work-sharing region upon the end of the work-sharing region, the side effects on other work-sharing constructs, described in 3.3.1, is reduced, because

dynamic threading has an influence on adaptive regions only. The flip side of dynamic threading is that at the end of an adaptive work-sharing loop, the threads of the finished region leave the team to help other adaptive regions, while there may still be work in the original team.
On the other hand, if threads would join another regions only upon the ending of the team, a parallel region following the just finished parallel region would maybe not have enough threads to start a team with the required size.

7.4 Conclusions

The presented dynamic threading extensions to the GNU Compiler Collection turned out to have a big potential for the exploiting of idle resources. The synthetic tests showed that big performance gains can be achieved while in most cases keeping the additional overhead low.
But there are a few cases in which the modified execution model performs worse. Therefore, propositions have been made in Section 7.2.2 to overcome the performance limitations. The implementation of the propositions could result in a solution to the deficiencies of nested OpenMP regions with irregular workloads and bring performance improvements for irregular nested parallel work-sharing loops.

A Appendix

A.1 OpenMP Example Application

Listing A.1 provides a simple example for an OpenMP program. At line 11 two parallel sections are created that are executed in parallel. The `private(th_id)` clause instructs the compiler, that `th_id` should be private to each thread. The first section goes from line 14 to 18 and the second section from line 21 to 33. The first section just outputs the thread number, by invoking the runtime library routine `omp_get_thread_num()` and terminates. Section two first also prints the thread number and then integrates from 0 to 2 over the function $y = x^2$. At line 27 the work-sharing loop that is actually doing the integration is started. That means the various iterations of this loop are divided among the available threads. The `reduction` clause is used to sum up the results of the parallel iterations. This result is then at the end of the second section also written to the console.

Listing A.1: Example of the usage of OpenMP.

```
1 #include <omp.h>
2 #include <stdio.h>
3
4 /* iteration steps */
5 #define NUM_STEPS (1<<29)
6
7 int main(int argc, char *argv[])
8 {
9   int th_id;
10 // Create two parallel sections
11 #pragma omp parallel sections private(th_id)
12 {
13 // First section just prints its thread number
14 #pragma omp section
15 {
16   th_id = omp_get_thread_num();
17   printf("Hello from thread: %d\n", th_id);
18 }
```

```
19 /* Second section integrates the function y=x^2
20  * from 0 to 2 in parallel */
21 #pragma omp section
22 {
23   th_id = omp_get_thread_num();
24   printf("Hello from thread: %d\n", th_id);
25   double x, sum=0;
26   int i;
27   #pragma omp parallel for reduction(+: sum)
28   for (i = 0; i<NUM_STEPS; i++) {
29     x = 2.0 * (double)i / (double)(NUM_STEPS); /* value of x */
30     sum += x * x / NUM_STEPS;
31   }
32   printf("Integration found result %f\n", sum);
33 }
34 }
35   return 0;
36 }
```

When compiled with OpenMP enabled, you can now execute this program and will get an output that looks similar to Listing A.2.

Listing A.2: Output of the OpenMP example program.

```
$ gcc -fopenmp example.c -o example
$ ./example
Hello from thread: 1
Hello from thread: 0
Integration found result 1.333333
```

A.2 libGOMP Structures

Listing A.3: Structures used to manage teams and workshares.

```
/* This structure contains the data to control one work-sharing
   construct, either a LOOP (FOR/DO) or a SECTIONS.  */

enum gomp_schedule_type
{
  GFS_STATIC,
  GFS_DYNAMIC,
  GFS_GUIDED,
```

```
  GFS_ADAPTIVE,
  GFS_RUNTIME
};

struct gomp_work_share
{
  /* This member records the SCHEDULE clause to be used for this
     construct. The user specification of "runtime" will already
     have been resolved. If this is a SECTIONS construct, this
     value will always be DYNAMIC.  */
  enum gomp_schedule_type sched;

  /* This is the chunk_size argument to the SCHEDULE clause.  */
  long chunk_size;

  /* This is the iteration end point.  If this is a SECTIONS
     construct, this is the number of contained sections.  */
  long end;

  /* This is the iteration step.  If this is a SECTIONS
     construct, this is always 1.  */
  long incr;

  /* This lock protects the update of the following members.  */
  gomp_mutex_t lock;

  union {
    /* This is the next iteration value to be allocated.  In the
       case of GFS_STATIC loops, this the iteration start point
       and never changes.  */
    long next;

    /* This is the returned data structure for SINGLE COPYPRIVATE.*/
    void *copyprivate;
  };

  /* This is the count of the number of threads that have exited
     the work share construct.  If the construct was marked nowait,
     they have moved on to other work; otherwise they're blocked
     on a barrier.  The last member of the team to exit the work
     share construct must deallocate it.  */
```

```
  unsigned threads_completed;

  /* This is the index into the circular queue ordered_team_ids of
     the current thread that's allowed into the ordered reason.  */
  unsigned ordered_cur;

  /* This is the number of threads that have registered themselves
     in the circular queue ordered_team_ids.  */
  unsigned ordered_num_used;

  /* This is the team_id of the currently acknowledged owner of
     the ordered section, or -1u if the ordered section has not
     been acknowledged by any thread.  This is distinguished from
     the thread that is *allowed* to take the section next.  */
  unsigned ordered_owner;

  /* This is a circular queue that details which threads will be
     allowed into the ordered region and in which order.  When a
     thread allocates iterations on which it is going to work, it
     also registers itself at the end of the array.  When a thread
     reaches the ordered region, it checks to see if it is the one
     at the head of the queue.  If not, it blocks on its RELEASE
     semaphore.  */
  unsigned *ordered_team_ids;
};

/* This structure contains all of the thread-local data
    associated with a thread team.  This is the data that must be
    saved when a thread
    encounters a nested PARALLEL construct.  */

struct gomp_team_state
{
  /* This is the team of which the thread is currently a member.*/
  struct gomp_team *team;

  /* This is the work share construct which this thread is currently
     processing.  Recall that with NOWAIT, not all threads may be
     processing the same construct.  This value is NULL when there
     is no construct being processed.  */
  struct gomp_work_share *work_share;
```

```
  /* This is the ID of this thread in the team.  This value is
     guaranteed to be between 0 and N-1, where N is the number of
     threads in the team.  */
  unsigned team_id;

  /* The work share "generation" is a number that increases by
     one for each work share construct encountered in the dynamic
     flow of the program.  It is used to find the control data
     for the work share when encountering it for the first time.
     This particular number reflects the generation of the
     work_share member of this struct.  */
  unsigned work_share_generation;

  /* For GFS_RUNTIME loops that resolved to GFS_STATIC, this is
     the trip number through the loop.  So first time a
     particular loop is encountered this number is 0, the second
     time through the loop is 1, etc.  This is unused when the
     compiler knows in advance that the loop is statically
     scheduled.  */
  unsigned long static_trip;
};

/* This structure describes a "team" of threads.  These are the
   threads that are spawned by a PARALLEL constructs, as well
   as the work sharing constructs that the team encounters.  */

struct gomp_team
{
  /* This lock protects access to the team data structures. */
  gomp_mutex_t lock;

  /* This is a dynamically sized array containing pointers to
     the control structs for all "live" work share constructs.
     Here "live" means that the construct has been encountered
     by at least one thread, and not completed by all threads.*/
  struct gomp_work_share **work_shares;

  /* The work_shares array is indexed by "generation &
     generation_mask". The mask will be 2**N - 1, where 2**N is
     the size of the array.  */
```

```
  unsigned generation_mask;

  /* These two values define the bounds of the elements of the
     work_shares array that are currently in use.  */
  unsigned oldest_live_gen;
  unsigned num_live_gen;

  /* This is the number of threads in the current team.  */
  unsigned nthreads;

  /* This is the saved team state that applied to a master
     thread before the current thread was created.  */
  struct gomp_team_state prev_ts;

  /* This barrier is used for most synchronization of the team.*/
  gomp_barrier_t barrier;

  /* Has the team already been started by the initial threads?
     adaptive threads should only join after the start. */
  bool started;

  /* This is the number threads from another team that are
     assisting the current team to execute an adaptive region.
     Before a team is ended the team ending thread has to wait
     until this becomes 0 again. */
  unsigned helpers;

  /* Team nesting depth. Needed for adaptive threads to know if
     they can join this team. They need to be on equal depth. */
  unsigned depth;

  /* This semaphore should be used by the master thread instead
     of its "native" semaphore in the thread structure. Required
     for nested parallels, as the master is a member of
     two teams.  */
  gomp_sem_t master_release;

  /* This array contains pointers to the release semaphore of
     the threads in the team.  */
  gomp_sem_t **ordered_release;
};
```

```
/* This structure contains all data that is private to libgomp
   and is allocated per thread.  */

struct gomp_thread
{
  /* This is the function that the thread should run upon launch. */
  void (*fn) (void *data);
  void *data;

  /* This is the current team state for this thread.  The ts.team
     member is NULL only if the thread is idle.  */
  struct gomp_team_state ts;

  /* This semaphore is used for ordered loops.  */
  gomp_sem_t release;

  /* Is this Thread a helper thread? This is only true when this
     thread belongs to an adaptive region and he has finished
     executing his original work_share and is now helping another
     adaptive work_share. */
  bool helper;

  /* This is the team_state of the current thread, before it
     became a helper. */
  struct gomp_team_state prev_ts;
};

/* ... and here is that TLS data.  */

#ifdef HAVE_TLS
extern __thread struct gomp_thread gomp_tls_data;
static inline struct gomp_thread *gomp_thread (void)
{
  return &gomp_tls_data;
}
#else
extern pthread_key_t gomp_tls_key;
static inline struct gomp_thread *gomp_thread (void)
{
  return pthread_getspecific (gomp_tls_key);
```

```
}
#endif
```

Listing A.4: Structures used for the function pool.

```
/* The relevant information about an adaptive region. */

struct fn_pool_entry
{
  /* Functionpointer to a loop code in an adaptive region. */
  void (*fn) (void *);

  /* The arguments for the function above. */
  void *data;

  /* The team_state of a thread that could need help. */
  struct gomp_team_state ts;

  /* The next entry in the fn_pool. */
  struct fn_pool_entry *next;
};


/* Global Pool of adaptive regions that are currently executed.
   It is needed to allow adaptive loop regions.   */

struct fn_pool_struct
{
  /* The actual pool - A list of fn_pool_entries. */
  struct fn_pool_entry *first, *last;

  /* This lock protects the update of the pool. */
  gomp_mutex_t lock;

};
```

Bibliography

[1] The GNU OpenMP library. http://gcc.gnu.org/projects/gomp/.

[2] The Message Passing Interface standard. http://www-unix.mcs.anl.gov/mpi/.

[3] The Omni OpenMP compiler. http://phase.hpcc.jp/Omni/.

[4] OmpSCR: The OpenMP source code repository. http://sourceforge.net/projects/ompscr/.

[5] The OpenMP specification, version 2.5. http://www.openmp.org/mp-documents/spec25.pdf.

[6] The OpenMP specification, version 3.0. http://www.openmp.org/mp-documents/spec30.pdf.

[7] The POSIX specification (IEEE std 1003.1,2004 edition). http://www.unix.org/version3/ieee_std.html.

[8] D. an Mey, S. Sarholz, and C. Terboven. Nested parallelization with OpenMP. *International Journal of Parallel Programming*, 35(5):459–476, 2007.

[9] E. Ayguadé, B. Blainey, A. Duran, J. Labarta, F. Martínez, X. Martorell, and R. Silvera. Is the schedule clause really necessary in OpenMP?, June 2003.

[10] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, October 2008.

[11] F. Broquedis, F. Diakhaté, S. Thibault, O. Aumage, R. Namyst, and P.-A. Wacrenier. Scheduling dynamic OpenMP applications over multicore architectures. *OpenMP in a New Era of Parallelism*, pages 170–180, 2008.

[12] J. Bull. Measuring synchonisation and scheduling overheads in OpenMP. In *First European Workshop on OpenMP*, Lund, Sweden, Oct. 1999.

[13] R. Buyya. *High Performance Cluster Computing: Programming and Applications*, volume 2, chapter 1. Prentice Hall PTR, 1999.

[14] N. Carriero, D. Gelernter, D. Kaminsky, and J. Westbrook. Adaptive parallelism with piranha. Technical report, 1993.

[15] M. Curtis-Maury, X. Ding, and C. D. Antonopoulos. An evaluation of OpenMP on current and emerging multithreaded multicore processors. In *In Proceedings of the First International Workshop on OpenMP*. Springer, June 2005.

[16] V. V. Dimakopoulos, P. E. Hadjidoukas, and G. C. Philos. A microbenchmark study of OpenMP overheads under nested parallelism. Technical report, Department of Computer Science, University of Ioannina, Ioannia, Greece, 2008.

[17] A. Duran. OpenMP 3.0: What's new? http://cobweb.ecn.purdue.edu/ParaMount/iwomp2008/documents/omp30.pdf.

[18] A. Duran, J. Corbalán, and E. Ayguadé. Evaluation of OpenMP task scheduling strategies. In *OpenMP in a New Era of Parallelism*. International Workshop on OpenMP 2008.

[19] A. Duran, J. M. Perez, E. Ayguadé, R. M. Badia, and J. Labarta. Extending the OpenMP tasking model to allow dependent tasks. In *OpenMP in a New Era of Parallelism*. International Workshop on OpenMP 2008.

[20] P. E. Hadjidoukas and V. V. Dimakopoulos. Nested parallelism in the OMPi OpenMP/C compiler. In *Euro-Par 2007 Parallel Processing*, August 2007.

[21] J. Merrill. GENERIC and GIMPLE: A new tree representation for entire functions. In *GCC Developers' Summit*, 2003.

[22] G. E. Moore. Cramming more components onto integrated circuits. *Electronics Magazine*, 38(8), Apr. 1965.

[23] D. Novillo. GCC an architectural overview, current status, and future directions. In *Proceedings of the Linux Symposium - Volume Two*, pages 185–200, Tokyo, Japan, Sept. 2006.

[24] D. Novillo. OpenMP and automatic parallelization in GCC. In *GCC Developers' Summit*, Ottawa, Canada, June 2006.

[25] J. Stokes. Pipelining: An overview, September 2004.

[26] M. A. Suleman, M. K. Qureshi, and Y. N. Patt. Feedback-driven threading: power-efficient and high-performance execution of multi-threaded workloads on CMPs. In *ASPLOS XIII: Proceedings of the 13th international conference on Architectural support for programming languages and operating systems*, pages 277–286, New York, NY, USA, 2008. ACM.

[27] H. Sutter. The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobb's Journal*, 30(3), Mar. 2005.

[28] X. Tian, A. Bik, M. Girkar, P. Grey, H. Saito, and E. Su. Intel OpenMP C++/Fortran compiler for hyper-threading technology: Implementation and performance. *Intel Technology Journal*, 6(1), Q1 2002.

[29] Y. Zhang, M. Burcea, V. Cheng, R. Ho, and M. Voss. An adaptive OpenMP loop scheduler for hyperthreaded SMPs. In D. A. Bader and A. A. Khokhar, editors, *Proceedings of the ISCA 17th International Conference on Parallel and Distributed Computing Systems (ISCA PDCS'04)*, pages 256–263, San Francisco, California, USA, Sept. 2004.